UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNCIA

TRABAJO FIN DE GRADO

Desarrollo de una plataforma para la interacción con la interfaz OBD-II utilizando BLE

Titulación: Grado en Ingeniería en Tecnologías de la Telecomunicación

Mención: Sistemas Electrónicos

Autor: D. Cristóbal Macías Cabrera

Tutores: D. Félix B. Tobajas Guerrero

D. Valentín De Armas Sosa

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Desarrollo de una plataforma para la interacción con la interfaz OBD-II utilizando BLE

HOJA DE EVALUACIÓN

Calificación: _____

Presidente

Fdo.: _____

Vocal **Secretario/a**

Fdo.: _____ Fdo.: _____

Fecha: Julio 2018

Contenido

Índice de figuras

Índice de tablas

Acrónimos

BLE	*Bluetooth Low Energy*
Bps	Bits por segundo
CAN	*Controller Area Network*
CCCD	*Client Characteristic Configuration Description*
CD-ROM	*Compact Disc Read-Only Memory*
CDMA/CR	*Code Division Multiple Access/Common receiver*
dBm	Decibelio-milivatio
DTC	*Diagnostic Trouble Codes*
ECU	*Engine Control Unit*
EDR	*Enhanced Data Rate*
EGR	*Exhaust Gas Recirculation*
GATT	*General Attribute Profile*
GPS	*Global Positioning System*
HW/SW	*Hardware/Software*
ID	Identificación
IDE	*Integrated Develpment Environment*
IEEE	*Institute of Electrical and Electronics Engineers*
IGIC	Impuesto General Indirecto Canario
ISO	*International Organization for Standardization*
ITU	*International Telecommunication Union*
IoT	*Internet of things*
KBPS	KiloBit por segundo
KWP	*Key World Protocol*
LED	*Light-Emitting Diode*
MAC	*Media Access Control*
MHz	*Mega Hertz*
Mbps	Megabits por segundo
OBD	*On Board Diagnostics*
OSI	*Open System Interconnection*
PCB	*Printed Circuit Board*
PDF	*Portable Document Format*
PID	*Parameter ID*
PWM	*Pulse With Modulation*

RGB	*Red,Green,Blue*
SAE	*Society of automotive Engineers*
SIG	*Special Interest Group*
SIM	*Subscriber Identify Module*
SPI	*Serial Peripheral Interface*
SSD	*Solid State Disk*
TFG	Trabajo Fin de Grado
UART	*Universal Asynchronous Reciever-Trasmitter*
ULPGC	Universidad de Las Palmas de Gran Canaria
UUID	*Universally Unique Identifier*
VPW	*Variable Pulse Width*

MEMORIA

1. Introducción

Internet de las Cosas (*IoT* por sus siglas en inglés, *Internet of Things)* comprende los procesos y aplicaciones necesarias para dotar de conectividad a dispositivos físicos y/o redes de sensores con el objetivo de construir una infraestructura de comunicación más eficiente y con una clara orientación al trabajo en la nube.

Además de basarse en sistemas de bajo consumo, los dispositivos IoT [1] se caracterizan por disponer de una elevada conectividad, ya sea mediante de puertos de entrada/salida de propósito general, tecnologías de comunicación inalámbrica *Wifi*, *Bluetooth*, etc.

Esta capacidad de interconexión de los dispositivos *IoT* puede cambiar radicalmente la manera en la que las personas interactúan con los diferentes dispositivos digitales e Internet [1]. Basándose en una conectividad parcialmente transparente para el usuario final, las capacidades de adaptación de la tecnología desarrollada bajo el concepto *IoT* son numerosas. Un buen ejemplo de la posible implementación de esta tecnología sería la posibilidad de desarrollar una nevera inteligente [2] con la capacidad de proporcionar al usuario información en tiempo real sobre las existencias de comida, la temperatura de la nevera, etc.

Las innovaciones proporcionadas por la tecnología *IoT* transformarán y potenciarán diferentes aspectos como la computación en la nube, la movilidad de dispositivos, o el procesamiento *big-data*. Esto, sin duda, modificará los paradigmas establecidos, tanto en entornos domésticos, como en entornos empresariales. Así, en estos entornos, doméstico y empresarial, se pueden identificar, entre otros, los siguientes ámbitos [3] de aplicación a partir de la integración de la tecnología *IoT*:

Empresarial:

- Dispositivos IoT industriales.
- Venta al por menor.
- Herramientas inteligentes y gestión de la energía.
- Sanidad.
- Ciudades inteligentes.

Doméstico:

- Hogares conectados.
- Dispositivos *IoT* personales.
- Coches conectados.
- Salud personal.

Dentro del entorno doméstico se encuentra la aplicación de la tecnología *IoT* en el ámbito de los coches conectados. En este Trabajo Fin de Grado se profundizará sobre las posibilidades que brinda la estandarización de los procesos industriales automovilísticos junto con las nuevas áreas de innovación en el campo de los dispositivos *IoT* conectados.

1.1 Antecedentes

Debido al creciente número de sensores y sistemas electrónicos implementados en los vehículos de combustión moderna, en 1983 la empresa Bosch desarrolló el protocolo CAN *(Controller Area Network).* Este nuevo protocolo permitiría crear un sistema de interconexión de dispositivos con un menor coste y cableado [4].

El puerto OBD-II *(On Board Diagnostics)* es la interfaz de comunicación que implementan los vehículos (a partir de 1998) para permitir la comunicación de un operario o sistema, con la centralita del vehículo o ECU *(Electronic Control Unit)* [5]. La centralita es un sistema centrado en el control de los sistemas eléctricos y electrónicos de los vehículos de combustión interna, destinado a la recopilación de información de los diferentes sensores y actuadores del vehículo. Fue presentado ante la SAE por Robert Bosch GmbH en febrero de 1986 [5].

Este protocolo estandarizado, ya presente actualmente en la mayoría de los vehículos de consumo general que circulan por las carreteras, permite ahondar en las posibilidades que ofrecen los diseños basados en dispositivos IoT con respecto a conectividad, escalabilidad y bajo coste.

Dentro de las capacidades de los dispositivos IoT en el ámbito de la automoción, se encuentran los sistemas de conducción autónoma asistidos por dispositivos de IoT,

coches conectados capaces de enviar información sobre el tráfico a otros vehículos en tiempo real, servicios de comunicaciones de emergencia, etc.

Otro problema añadido a los sistemas actuales, que los diferencian de la filosofía de los dispositivos IoT, es el coste. Gracias a estos pequeños dispositivos es posible crear sistemas con una alta escalabilidad asumiendo unos bajos costes, lo cual contribuye al impulso que esta tecnología experimenta actualmente.

Llamada a ser la "siguiente revolución industrial" [3], se prevé que los dispositivos IoT conectarán alrededor de 20 billones de dispositivos inteligentes a la actual infraestructura de Internet. En la Figura 1 se puede apreciar el crecimiento de los dispositivos IoT frente al número de personas. A finales de 2016 se observa cómo se alcanza el punto en el que ya existen más dispositivos conectados, que personas en el mundo. Para el año 2020 la previsión refleja una estimación de 20 billones de dispositivos.

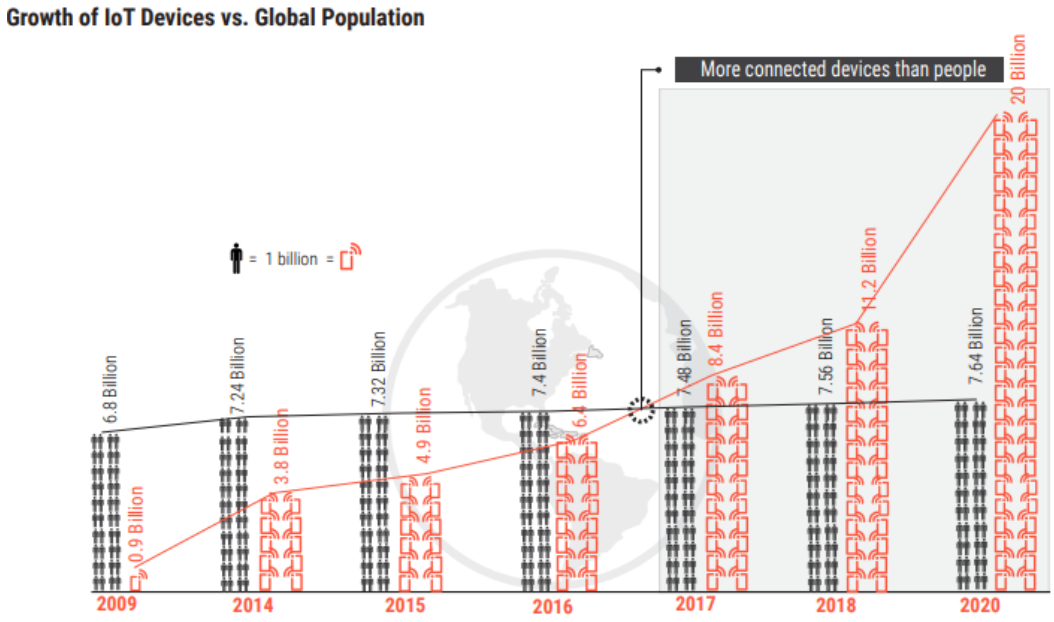

Figura 1. Gráfica crecimiento personas/dispositivos IoT

En este Trabajo Fin de Grado (TFG) se hará hincapié en las posibilidades de conectividad y compatibilidad que ofrecen estos dispositivos de IoT para crear una

plataforma HW/SW de bajo coste que posibilite la comunicación entre la centralita de un vehículo y el usuario mediante tecnología *Bluetooth Low Energy* (BLE).

1.2 Peticionario

Actúa como peticionario del presente Trabajo Fin de Grado (TFG) la Escuela de Ingeniería de Telecomunicación y Electrónica (EITE) de la Universidad de Las Palmas de Gran Canaria (ULPGC) como requisito indispensable para la obtención del título de Graduado en Ingeniería en Tecnologías de la Telecomunicación, tras haber superado con éxito las asignaturas especificadas en el Plan de Estudios.

1.3 Objetivos del trabajo

El objetivo principal de este TFG consiste en la creación de una plataforma HW/SW que sea capaz de extraer la información que un vehículo proporciona a través del puerto OBD-II, y transmitir esta información mediante una pasarela BLE, implementada en este caso en el dispositivo *RedBear Duo* de la empresa RedBear, con el fin de establecer la comunicación con un terminal móvil, mediante el protocolo *Bluetooth Low Energy*.

En las fases iniciales del proyecto se utilizará el dispositivo *Photon* de la empresa Particle (fundamentalmente por motivos de robustez), que mediante el adaptador proporcionado por el kit de desarrollo *Carloop* y su conectividad Wi-Fi, permitirá la comunicación entre la centralita del vehículo que contiene toda la información de los registros de sucesos y captadores, con el dispositivo de IoT. Este dispositivo carece de conectividad BLE, pero el *firmware* desarrollado para éste es altamente compatible con el dispositivo final *RedBear Duo*, que sí dispone de esta conectividad.

Antes de la implementación sobre el dispositivo final *RedBear Duo*, se procederá a realizar una verificación inicial del funcionamiento del *firmware* desarrollado con el dispositivo *Photon*. Esto proporcionará una garantía de funcionamiento que permitirá un mejor seguimiento del TFG. Esta verificación consistirá en la creación de una pasarela entre el dispositivo *Photon* y el terminal móvil que permite, utilizando la conectividad *Wi-*

Fi del dispositivo y su interfaz serie, una comunicación bidireccional con el usuario para comprobar el correcto funcionamiento del *firmware*.

Finalmente, se implementará en el dispositivo *RedBear Duo* la funcionalidad completa de un dispositivo *Peripheral* BLE para poder establecer una conexión con un dispositivo móvil que permita la transferencia de las órdenes/datos que se enviarán/recibirán a través de la interfaz OBD-II, una vez integrado en la plataforma HW/SW final el código inicialmente desarrollado.

1.4 Estructura de la memoria

El presente documento se ha dividido en cuatro partes diferenciadas: *Memoria*, *Pliego de condiciones*, *Presupuesto* y *Anexo*. A su vez, la *Memoria* se ha estructurado en cinco capítulos, tal como se describe a continuación:

La memoria se ha estructurado de manera que corresponda al desarrollo seguido para la realización de este Trabajo Fin de Grado.

Así, una vez introducido el trabajo en este primer capítulo, en el segundo capítulo "Planteamiento inicial de la plataforma HW/SW**",** se analizarán las condiciones expuestas para la realización de este trabajo, de manera que se forme una estructura clara del problema a resolver y la metodología que se ha seguido para resolverlo. Además, en este capítulo se analizarán en profundidad los componentes hardware utilizados, haciendo hincapié en las características que atañen al correcto desarrollo del *firmware*. Se hará también un análisis de las herramientas *software* necesarias para desarrollar el *firmware* de manera satisfactoria.

Tras el análisis de los medios que se utilizarán, en el capítulo tres "Implementación de la interrogación vía serie" se presentarán las condiciones de desarrollo de la implementación inicial de la pasarela mediante el dispositivo *Photon* y las características o limitaciones encontradas de cara a la implementación utilizando el dispositivo *RedBear Duo.* Una vez completado el desarrollo inicial del *firmware* y realizada la tarea de corrección y validación, se obtendrá un punto de partida para la posterior implementación en la plataforma *HW/SW* final.

En el capítulo cuatro “Implementación de la interrogación vía BLE”, partiendo del análisis de la tecnología BLE que se utilizará para la plataforma HW/SW final, se realizará el desarrollo del *firmware* para el dispositivo *RedBear Duo* incluyendo la totalidad de las funciones planteadas en los objetivos de este trabajo. Además, se realizará la corrección y validación de la pasarela final BLE para comprobar su correcto funcionamiento.

A continuación, en el capítulo “conclusiones” se realizará una valoración de los resultados obtenidos a partir de la realización del presente TFG, y sus implicaciones con respecto a las posibles líneas futuras del trabajo.

Finalmente, se presentará la bibliografía utilizada en este Trabajo Fin de Grado.

Por su parte, en el *Pliego de condiciones* se expondrán las condiciones bajo las que se ha desarrollado el presente TFG, mientras que en el *Presupuesto* se recogerán los gastos generados en la realización del trabajo, y en el *Anexo* se incluirá la estructura y el contenido del CD-ROM adjunto, además de toda la información que finalmente se incluya.

2. Planteamiento inicial de la plataforma HW/SW

Como se adelantaba en el primer capítulo de este documento, los vehículos de combustión interna modernos -a partir de 1998- [6] implementan en su sistema de control electrónico el robusto bus de comunicación CAN. Este trabajo se apoya en esta característica para el desarrollo del *firmware* de los dispositivos IoT con el fin de poder realizar comunicaciones bidireccionales con la centralita del vehículo.

2.1 Protocolo BUS CAN

El protocolo CAN *(Controller Area Network)* fue diseñado por Bosch en 1986 [4], originalmente ideado con el fin de reducir la cantidad de cableado necesario, se ha convertido en un estándar de utilización en automóviles modernos. Actualmente este protocolo se utiliza como un bus multi-maestro para conectar todo tipo de dispositivos inteligentes. CAN fue estandarizado en 1993 como ISO 11898-1, un protocolo que solo define hasta la capa dos del modelo de referencia OSI *(Open System Interconnection)* representado en la Figura 2. Este hecho, junto con las características propias del bus, han permitido una proliferación de diferentes protocolos basados en CAN modificando las diferentes capas del estándar OSI.

Figura 2. Capas modelo OSI

Según los niveles o capas propuestas por el estándar OSI, el protocolo CAN solo define los dos primeros niveles (Capa Física y Capa de Enlace). Partiendo del estándar propuesto, diferentes fabricantes han amoldado su sistema de comunicación basado en CAN para obtener un rendimiento óptimo en sus procesos de fabricación.

En este Trabajo Fin de Grado se abordará el uso de este protocolo en el marco de vehículos modernos, en los cuales, el bus de comunicación CAN está orientado a la gestión de todos los sensores y captadores del propio vehículo, representados como referencia en la Figura 3.

Figura 3. Ejemplo cableado interno vehículo. (Bentley)

2.1.1 Características del protocolo de comunicación

El protocolo CAN está basado en un sistema productor/consumidor asíncrono donde cada nodo está siempre a la escucha, y las comunicaciones están controladas por un dispositivo que hará de árbitro en el bus. Las peticiones de información se crean de acuerdo con una tabla de órdenes predefinidas que contienen los identificadores de las variables.

Una vez decodificado el nombre de la variable asociada al nodo al que realiza la petición, éste envía la información actualizada. Todos los nodos, incluidos el transmisor de la información solicitada, permanecen activos mientras haya actividad en el bus.

De manera característica, todos los nodos deben recibir el mensaje enviado desde el árbitro del bus y aceptarlo, en caso contrario, se presupone un error en la transmisión.

CAN utiliza un método de acceso al bus por prioridades mediante la técnica CDMA/CR *(Carrier Sense Multiple Access/ Collision Resolution),* que resuelve los conflictos de acceso

al bus mediante técnicas que no destruyen la información a transmitir. Los nodos no disponen de direcciones físicas, por lo que los nodos reciben todos los mensajes y será tarea de cada uno de ellos revisar el identificador, resaltado en color amarillo en las estructuras de una Trama CAN normal y extendida, mostradas en la Figura 4

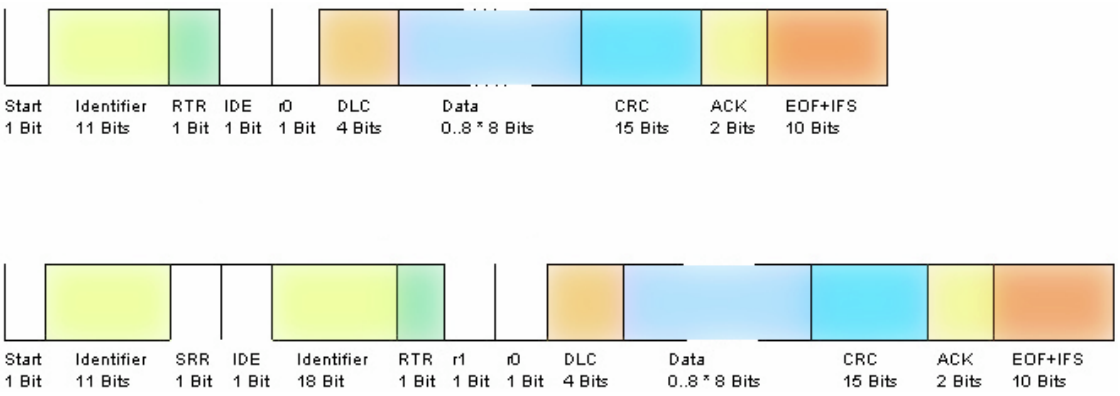

Figura 4. Trama CAN normal y extendida

En caso de trabajar con la versión estándar compacta de la trama CAN, cada nodo buscará su identificador para resolver la petición [4], cuyo tamaño máximo es de 11 bits en la versión compacta, y 29 de bits en la versión extendida de la trama.

2.1.2 Medio físico del bus CAN

El medio físico del bus CAN consiste en un cable de par trenzado con los terminales especificados. Así, en la especificación básica de CAN, el bus puede tener un máximo de 32 nodos. El número de mensajes por segundo varía entre 2000 y 5000, a una velocidad de 250kbps (velocidad máxima para la especificación básica de CAN en modo compacto) y dependiendo del número de bytes por mensaje. Para la versión extendida de CAN, este límite se encuentra en 1 Mbps [8].

El par trenzado que interconecta los distintos nodos de un bus CAN debe tener una impedancia característica de 120Ω. Además, el estándar no especifica un conector para el bus, por lo que cada aplicación puede disponer de un conector distinto. No obstante, existen ciertos formatos comúnmente aceptados, como el conector D-Sub.

Los hilos que componen el par trenzado reciben el nombre de CAN_H y CAN_L. Como se representa en la Figura 5, en estado inactivo la diferencia de tensión entre ambos es de 2.5 V. Para enviar un “1” lógico se establece el hilo CAN_H a más tensión que CAN_L, mientras que un “0” lógico se envía estableciendo un voltaje superior en la línea CAN_L. El uso de tensiones diferenciales aporta robustez al sistema frente a ambientes muy ruidosos o daños en las líneas.

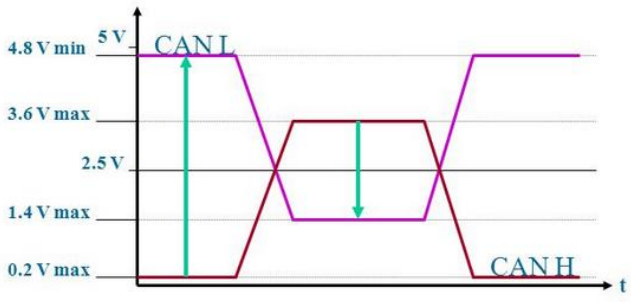

Figura 5. Ejemplo señales del trenzado CAN

2.2 Interfaz de diagnóstico OBD II

OBD *(On-board diagnostics)* es el término que hace referencia a la capacidad implementada en los vehículos para autodiagnóstico y verificación mediante el uso de un conector estandarizado [9], mostrado en la Figura 6. En un inicio, este sistema solo podía alertar de errores encendiendo una luz indicadora genérica, pero esta situación cambió en versiones posteriores del estándar, así como la creación de una serie de códigos de errores estandarizados, lo que ha permitido su rápida implementación en los vehículos modernos.

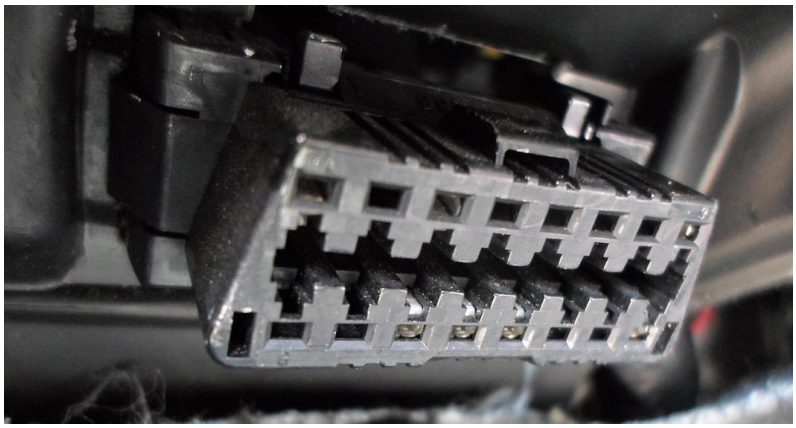

Figura 6. Ejemplo conector OBD-II

En los últimos años, cada fabricante ha desarrollado su propio protocolo de comunicación utilizando el conector estándar OBD-II con diferentes buses de comunicación (CAN, VPW, PWM, ISO, KWP) [9]. En concreto, el presente TFG se centrará en el estándar SAE J1979 que se basa en el uso de identificadores de parámetros estandarizados (*Parameter IDs,* PID) dentro del bus de comunicación CAN que incorporan los vehículos de manera genérica a partir de 2008.

Definido por el estándar SAE J1979, un técnico puede, mediante un dispositivo de diagnóstico como el que se muestra en la Figura 7, acceder a estos parámetros creando peticiones en el bus. Estas peticiones tienen un formato definido que el vehículo analizará, generando en cada caso la respuesta correspondiente.

Figura 7. Herramienta de diagnóstico OBD II

Debido a la propia arquitectura del bus CAN, la petición creada irá dirigida a todos los nodos del bus, siendo el dispositivo objetivo de la petición el único que responderá introduciendo la respuesta en el bus.

2.2.1 Modos de operación

El estándar SAE J1979 define diez modos de operación [9] para el sistema de diagnóstico OBD-II. Cada uno de ellos tiene la utilidad que se detalla en la Tabla 1.

Tabla 1. Modos de operación OBD-II (Estándar SAE J1979)

Modo(HEX)	Descripción.
01	Mostrar información.
02	Mostrar datos de cuadros congelados.
03	Mostrar códigos de error almacenados.
04	Limpiar códigos de errores y valores almacenados.
05	Resultados de test, monitorización de sensores de oxígeno.
06	Resultados de test, monitorización de otros componentes.
07	Mostrar códigos de error pendientes.
08	Control de sistemas de diagnóstico a bordo.
09	Petición de información del vehículo.
0A	Códigos de error permanentes.

Para la realización de este Trabajo Fin de Grado, se ha seleccionado el modo de operación 01, ya que el objetivo de la plataforma HW/SW final a desarrollar es la transmisión de información desde la centralita del vehículo, hacia el dispositivo de usuario con conectividad BLE.

2.2.2 Formato de petición y respuesta

Las peticiones y respuestas del vehículo tienen lugar a través del bus CAN de comunicación utilizando direcciones para acceder a cada dispositivo. El dispositivo encargado de realizar la petición al bus lanzará en primer lugar una petición genérica con un ID de valor 0x7DF. Esta dirección genérica se comporta como una dirección *broadcast*. La centralita del vehículo puede responder a cualquier dirección incluida en el rango 0x7E0 - 0x7E7, y procederá a generar una respuesta añadiendo el valor 8 a la dirección utilizada para la primera petición. Como referencia, si se realiza una petición al bus CAN con ID 0x7EF, la respuesta que se obtendrá con ID 0x7E8 [9].

Debido a que el sistema utiliza el bus CAN, las transmisiones de información útil quedan limitadas a un máximo de 8 *bytes*.

Petición

Según el estándar SAE J1979, la sección dedicada a los datos útiles en una trama de petición (8 *bytes*) debe seguir el formato representado en la Tabla 2.

Tabla 2. Formato trama petición

Byte	0	1	2	3	4	5	6	7
Función	Número de *bytes* adicionales	Modo de operación	Código PID	Sin uso				

Respuesta

Una vez realizada la petición a la centralita del vehículo, esta responderá con una trama, también definida por el estándar SAE J1979, con el formato mostrado en la Tabla 3.

Tabla 3. Formato trama respuesta

Byte	0	1	2	3	4	5	6	7
Función	Número de *bytes* adicionales	Modo de operación	Código PID	Byte de información 1	Byte de información 2	Byte de información 3	Byte de información 4	Sin uso

Observando la Tabla 3, los *bytes* que representan la información que se ha solicitado se corresponden con los *bytes* 3, 4, 5 y 6 de la trama de respuesta. Esta información viene codificada según el estándar SAE J1979, analizándose en el siguiente apartado la codificación de la información

Cuando se realiza una petición a una dirección PID específica -por ejemplo, las revoluciones del motor-, la trama de respuesta que se obtendrá en el sistema de diagnóstico incluirá, en el byte 2, el mismo código PID enviado en la petición. Por su parte, los bytes 3, 4, 5 y 6 serán los encargados de transportar la información [9]. Cabe destacar que el número de bytes de respuesta está determinado por el estándar. En caso de que el PID seleccionado no requiera el uso de todos los bytes de la trama, dejará a 0x00 aquellos que no utilice.

2.2.3 Direcciones de PID estandarizados

Según la definición de SAE, existe una disposición de PID ya estandarizados. Estos PID están asociados a los bits de respuesta que proporciona el vehículo cuando se le realizan peticiones de información a través del bus. Los formatos de la trama de petición y de respuesta son los presentados en el apartado 2.2.2 de este documento.

En concreto, el modo 1 de operación -el que se utiliza en este TFG- existen cuatro PID específicos [10] que permiten determinar la disponibilidad, de un determinado rango de PID previamente definido. Esta utilidad se basa en el uso de un conjunto de PID cuya única función es la de informar acerca de la disponibilidad de los demás PID del vehículo. Esto permite al sistema de diagnóstico conocer qué PID están disponibles en cada vehículo para realizar peticiones.

Así, como referencia, al realizar una petición de información al primero de estos PID, se obtendrá como respuesta 4 *bytes* que corresponden a un *bitmap* de 32 *bits*. Este *bitmap* representa mediante un "1" lógico aquellas posiciones correspondientes a las direcciones de PID disponibles comprendidas entre 0x01 y 0x1F, y un "0" lógico en caso contrario.

Siguiendo la estructura propuesta por el párrafo anterior, un vehículo puede disponer de hasta cuatro PID asociados a estos *bitmaps* de disponibilidad. Las direcciones de estos PID son: 0x00 (primer *bitmap* de disponibilidad), 0x20 (segundo *bitmap* de disponibilidad),

0x40 (tercer *bitmap* de disponibilidad) y 0x60 (cuarto y último *bitmap* de disponibilidad). En la Tabla 4 se indica la relación entre los PID de disponibilidad y los rangos de PID disponibles que incluyen.

Tabla 4. Rangos de disponibilidad devuelto por las direcciones de PIDS

Dirección	Rango de disponibilidad
0x00 (Primer *bitmap)*	0x01-0x20
0x20 (Segundo *bitmap*)	0x21-0x40
0x40 (Tercer *bitmap*)	0x41-0x60
0x60 (Cuarto *bitmap*)	0x61-0x80

En la Tabla 5 se muestra un ejemplo de *bitmap* de respuesta al realizar una petición al PID con dirección 0x00.

Tabla 5. Bitmap ejemplo respuesta primer PID disponibilidad

Hexadecimal	**F**				**F**				**0**				**0**				**A**				**B**				**E**				**1**			
Binario	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	1	1	0	0	0	0	1

En la Tabla 5 se aprecia un ejemplo de la respuesta que se podría recibir al realizar una petición al PID con dirección 0x00, que se corresponde con el primer PID de disponibilidad. En este caso, la información de respuesta obtenida se corresponde a la palabra en hexadecimal 0xFF00ABE1. La correspondencia del *bitmap* binario con respecto a las direcciones PID soportadas es correlativa, de forma que el primer valor en hexadecimal "F" corresponde al código binario "1111", lo cual indica que las direcciones de PID 0x01, 0x02, 0x03 y 0x04 están disponibles. De manera análoga, los PID disponibles correspondientes al último valor recibido (0x1) corresponde a las direcciones 0x1D, 0x1E, 0x1F y 0x20, de las cuales se indica que únicamente se encuentra disponible la dirección 0x20.

En este trabajo solo se han utilizado los tres primeros *bitmaps* de disponibilidad, ya que son los que el vehículo de pruebas tiene disponible. En la Tabla 6, la Tabla 7 y la Tabla 8 se relacionan las direcciones de los PID del vehículo -según el estándar SAE- con su descripción.

Tabla 6. Lista PID estándar primer bitmap

PID (HEX)	PID (DEC)	Número de *bytes* de respuesta	Descripción
00	0	4	PIDs soportados [0x01 - 0x20]
01	1	4	Estado de monitorización desde la última limpieza de errores.
02	2	2	Congelar DTC (Diagnostic Trouble Codes).
03	3	2	Estado del sistema de combustible.
04	4	1	Carga del motor calculada.
05	5	1	Temperatura del líquido refrigerante.
06	6	1	Ajuste de combustible a corto plazo. (banco 1)
07	7	1	Ajuste de combustible a largo plazo. (banco 1)
08	8	1	Ajuste de combustible a corto plazo. (banco 2)
09	9	1	Ajusto de combustible a largo plazo. (banco 2)
0A	10	1	Presión de combustible.
0B	11	1	Presión absoluta en el colector de entrada.
0C	12	2	Revoluciones del motor.
0D	13	1	Velocidad del vehículo
0E	14	1	Temporización de avance. (punto muerto superior)
0F	15	1	Temperatura del aire de entrada
10	16	2	Flujo de aire. (caudalímetro)
11	17	1	Posición del acelerador
12	18	1	Estado del sistema secundario de aire.
13	19	1	Sensores de oxígeno presentes (en dos bancos)
14	20	2	Sensor 1 de oxígeno.
15	21	2	Sensor 2 de oxígeno.
16	22	2	Sensor 3 de oxígeno.
17	23	2	Sensor 4 de oxígeno.
18	24	2	Sensor 5 de oxígeno.
19	25	2	Sensor 6 de oxígeno.
1A	26	2	Sensor 7 de oxígeno.
1B	27	2	Sensor 8 de oxígeno.
1C	28	1	Estándar OBD que utiliza el vehículo
1D	29	1	Sensores de oxígeno presentes (en 4 bancos)
1E	30	1	Estado de la entrada auxiliar
1F	31	2	Tiempo desde que el motor esta encendido.

Tabla 7. Lista PIDs estándar segundo bitmap

PID (HEX)	PID (DEC)	Número de *bytes* de respuesta	Descripción
20	32	4	PIDs soportados [0x21 - 0x40]
21	33	2	Distancia recorrida con luz de malfunción encendida.
22	34	2	Presión de inyección directa.
23	35	2	Calibre de la presión de inyección.
24	36	4	Sensor de oxígeno 1.
25	37	4	Sensor de oxígeno 2.
26	38	4	Sensor de oxígeno 3.
27	39	4	Sensor de oxígeno 4.
28	40	4	Sensor de oxígeno 5.
29	41	4	Sensor de oxígeno 6.
2A	42	4	Sensor de oxígeno 7.
2B	43	4	Sensor de oxígeno 8.
2C	44	1	Válvula EGR
2D	45	1	Error en válvula EGR
2E	46	1	Purga de evaporación
2F	47	1	Porcentaje de entrada del tanque de combustible
30	48	1	Calentamientos desde limpieza de errores.
31	49	2	Distancia recorrida desde limpieza de códigos.
32	50	2	Presión del sistema de evaporación
33	51	1	Presión barométrica absoluta.
34	52	4	Sensor de oxígeno 1.
35	53	4	Sensor de oxígeno 2.
36	54	4	Sensor de oxígeno 3.
37	55	4	Sensor de oxígeno 4.
38	56	4	Sensor de oxígeno 5.
39	57	4	Sensor de oxígeno 6.
3A	58	4	Sensor de oxígeno 7.
3B	59	4	Sensor de oxígeno 8.
3C	60	2	Temperatura catalizador. (Banco 1, sensor 1)
3D	61	2	Temperatura catalizador. (Banco 1, sensor 2)
3E	62	2	Temperatura catalizador. (Banco 2, sensor 1)
3F	63	2	Temperatura catalizador. (Banco 2, sensor 2)

Tabla 8. Lista PIDs estándar tercer bitmap

PID (HEX)	PID (DEC)	Número de *bytes* de respuesta	Descripción
40	64	4	PIDs soportados [0x41 -0x60]
41	65	4	Estado de monitorización en ciclo actual
42	66	2	Módulo de control de voltaje.
43	67	2	Valor de carga absoluto.
44	68	2	Relación combustible-aire
45	69	1	Posición relativa del acelerador.
46	70	1	Temperatura ambiente.
47	71	1	Posición absoluta acelerador B
48	72	4	Posición absoluta acelerador C
49	73	4	Posición acelerador D.
4A	74	4	Posición acelerador E.
4B	75	4	Posición acelerador F.
4C	76	1	Actuador acelerador
4D	77	2	Tiempo con luz de aviso encendida.
4E	78	2	Tiempo desde limpieza de errores
4F	79	4	Relación máxima aire-combustible, voltaje del sensor de oxígeno, corriente del sensor de oxígeno y presión absoluta del colector de entrada.
50	80	4	Valor máximo de salida para flujo de aire desde caudalímetro
51	81	1	Tipo de combustible
52	82	1	Porcentaje de etanol en el combustible
53	83	2	Presión absoluta del sistema de evaporación
54	84	2	Presión del sistema de evaporación
55	85	2	Ajuste mezcla a corto plazo sensor de oxígeno secundario (banco 1 y banco 3)
56	86	2	Ajuste mezcla a largo plazo sensor de oxígeno secundario (banco 1 y banco 3)
57	87	2	Ajuste mezcla a corto plazo sensor de oxígeno secundario (banco 2 y banco 4)
58	88	2	Ajuste mezcla a largo plazo sensor de oxígeno secundario (banco 2 y banco 4)
59	89	2	Presión del sistema de inyección de combustible
5A	90	1	Posición relativa del acelerador

5B	91	1	Vida útil restante de la batería (sistemas híbridos)
5C	92	1	Temperatura del aceite del motor
5D	93	2	Temporalización de la inyección de combustible
5E	94	2	Consumo de combustible
5F	95	1	Requisitos de emisiones a los que el vehículo se ajusta

2.3.4 Decodificación de las tramas de respuesta.

Como se ha indicado en el apartado 2.2.2 “Formato de petición y respuesta”, la trama de respuesta del vehículo se compone de 8 bytes, de los cuales solo 4 bytes corresponden a la información correspondiente al PID asociado a la petición. El estándar define un formato específico para la correcta interpretación de la información que envía el vehículo. Además de incluir la información en crudo, la trama de respuesta incluye un byte con el código PID utilizado en la petición, lo cual servirá de ayuda en el presente TFG para el desarrollo del *firmware* de usuario en los dispositivos IoT.

Debido a que ciertas direcciones PID utilizan una codificación de la respuesta a nivel de bit [10], se utilizará la nomenclatura reflejada en la Tabla 9 para clarificar su decodificación.

Tabla 9. Nomenclatura a nivel de bit de la respuesta del vehículo

Byte información 1								**Byte información 2**								**Byte información 3**								**Byte información 4**							
A								**B**								**C**								**D**							
A7	**A6**	**A5**	**A4**	**A3**	**A2**	**A1**	A0	**B7**	**B6**	**B5**	**B4**	**B3**	**B2**	**B1**	B0	**C7**	**C6**	**C5**	**C4**	**C3**	**C2**	**C1**	C0	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	D0

En el Anexo I se recoge la decodificación de la información enviada como respuesta por parte del vehículo tras haber sido realizada una petición a un PID.

2.3Arquitectura de la plataforma HW/SW inicial

Como se ha mencionado en capítulos anteriores, el primer acercamiento que se ha realizado en el desarrollo de este TFG ha consistido en la implementación de la interconexión de la pasarela utilizando el dispositivo *Photon*. En este apartado se presenta la forma en la que se ha realizado dicha interconexión, así como las diferencias y/o similitudes con respecto a la pasarela HW/SW final en la que se integra el dispositivo *RedBear Duo*.

2.3.1 Plataforma HW/SW basada en el dispositivo *Photon*

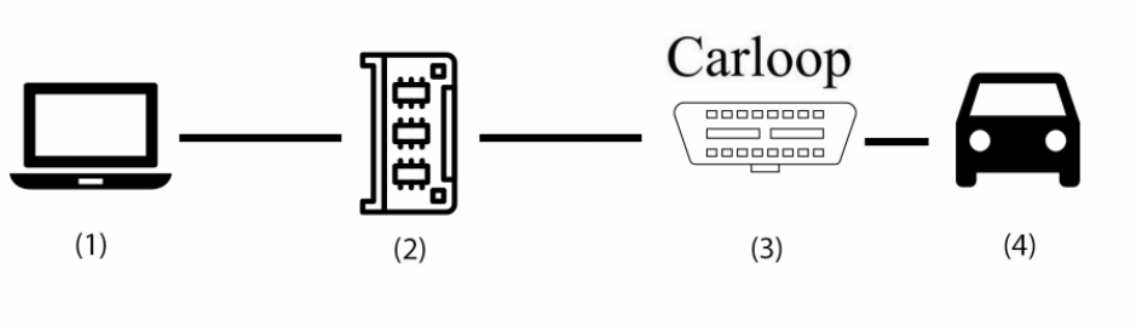

Figura 8. Esquema conexionado con dispositivo Photon.

(1) Portátil, (2) Dispositivo IoT Photon, (3) Interfaz OBD-II Ca, (4) Conexión con Bus CAN del vehículo

El esquema propuesto por la Figura 8 es el que se ha utilizado en la primera fase de desarrollo del *firmware*. El ordenador portátil está conectado por medio del puerto serie al dispositivo *Photon*, que a su vez está conectado al vehículo por medio de la interfaz *Carloop* y el puerto OBD-II. Esto permitirá utilizar la conectividad Wi-Fi del dispositivo *Photon* para la programación del *firmware*, y su conectividad serie para realizar las labores de interfaz de usuario y validación funcional.

2.3.2 Plataforma HW/SW inicial basada en el dispositivo *RedBear Duo*

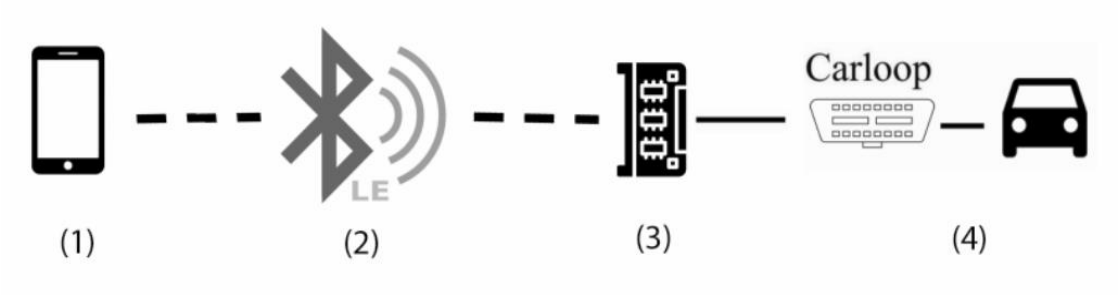

Figura 9. Esquema conexionado plataforma HW/SW final.

(1) Dispositivo móvil, (2) Conectividad Bluetooth, (3) Dispositivo RedBear Duo, (4) Interfaz *Carloop* - vehículo

Siguiendo un esquema similar al propuesto para el dispositivo *Photon*, para la plataforma final se hará uso del dispositivo *RedBear Duo*, como se representa en la Figura 9. Éste, gracias a su conectividad *Bluetooth Low Energy*, permitirá crear una pasarela de comunicación con un terminal dotado de esta tecnología. En este trabajo se ha utilizado un teléfono móvil que implementa este protocolo de comunicación.

Esta pasarela permitirá realizar comunicaciones bidireccionales entre el dispositivo *RedBear Duo*, que a su vez está interconectado el vehículo mediante la interfaz *Carloop*.

2.4 Compomentes hardware

2.4.1 Ordenador portátil

El dispositivo utilizado para el desarrollo del código correspondiente al *firmware* de los dispositivos IoT es un portátil *Macbook Pro* de la marca Apple [11], mostrado en la Figura 10.

Figura 10. Ordenador portátil

La conectividad USB y Wi-Fi, unida a la compatibilidad para el desarrollo del *firmware* de los dispositivos, permite realizar de manera correcta todas las tareas *software* necesarias para este trabajo.

2.4.2 Terminal móvil

Para el desarrollo del *firmware* es necesaria la utilización de un dispositivo con conectividad BLE *(Bluetooth Low Energy)* que actúe como dispositivo *Master* en la plataforma HW/SW final. El dispositivo con esta conectividad utilizado es el *smartphone* Huawei Mate 10 [12], representado en la Figura 11.

Figura 11.Terminal móvil con conectividad BLE

Este terminal se basa en un procesador Huawei Kirin 970, e incluye 4 GB de RAM utilizando el sistema operativo Android 8 Oreo.

2.4.3 Vehículo de pruebas

El vehículo utilizado para la validación funcional de la plataforma desarrollada en el presente TFG ha sido el Hyundai ix35 [13], representado en la Figura 12, en su versión Gasolina.

Figura 12. Hyundai ix35

Este vehículo dispone del conector de diagnóstico OBD-II, que se encuentra en la parte inferior del volante, y más concretamente en la zona de los pedales. Este conector OBD-II permitirá la interacción con el bus de comunicación CAN que este implementa.

2.4.4 Dispositivo *Photon*

Photon es el nombre que recibe el dispositivo de IoT de la empresa Particle, mostrado en la Figura 13. Este producto está concebido como un pequeño microcontrolador con un sistema operativo en tiempo real (*FreeRTOS*) orientado al desarrollo de soluciones inteligentes de bajo coste y alta compatibilidad entre sistemas [14].

Basado en la arquitectura *WICED* de la empresa *Cypress*, incorpora un microcontrolador STM32F205 ARM Cortex-M3 con una señal reloj de frecuencia 120 MHz, y un módulo Wi-Fi BCM43362 de Broadcom. Además, incluye 1 MB de memoria *flash* para almacenar el *firmware*, y 128 KB de memoria RAM para la ejecución.

Está caracterizado por trabajar en la banda de Wi-Fi de 2.4GHz con el protocolo IEEE 802.11b/g/n ofreciendo tasas de transferencia de hasta 65 Mbit/s. Dispone, asimismo, de diferentes modos de operación, entre los que se encuentran: "*stand-by*", "*stop mode*" y "*ultra low power sleep*"

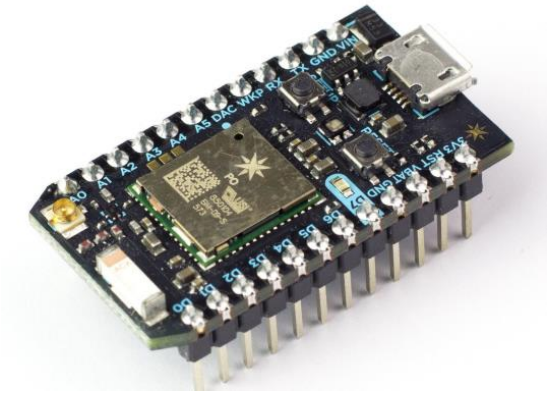

Figura 13. Dispositivo Photon

Para la aplicación de este dispositivo en el presente TFG, se utilizará adicionalmente la interfaz que proporciona el módulo *Carloop,* que permitirá la interconexión entre los pines del dispositivo *Photon* y los pines del puerto de diagnóstico OBD II.

Entre otras características, el dispositivo *Photon* dispone de 1MB de memoria *Flash* y 128KB de RAM, además de varios LED (*Light-Emitting Diode*) integrados, 18

entradas/salidas de propósito general, periféricos avanzados y un sistema operativo en tiempo real (*FreeRTOS*). Este dispositivo cuenta con una gran cantidad de interfaces analógicas, digitales y de comunicación, tal y como se muestra en la Tabla 10.

Tabla 10. Periféricos del dispositivo Photon

Tipo de Periférico	Cantidad	Entrada / Salida
Digital	18	Entrada / Salida
Analógico (ADC)	8	Entrada
Analógico (DAC)	2	Salida
SPI	2	Entrada / Salida
I2S	1	Entrada / Salida
I2C	1	Entrada / Salida
CAN	1	Entrada / Salida
USB	1	Entrada / Salida
PWM	9	Salida

En la Figura 14 se muestran los principales componentes del dispositivo *Photon*.

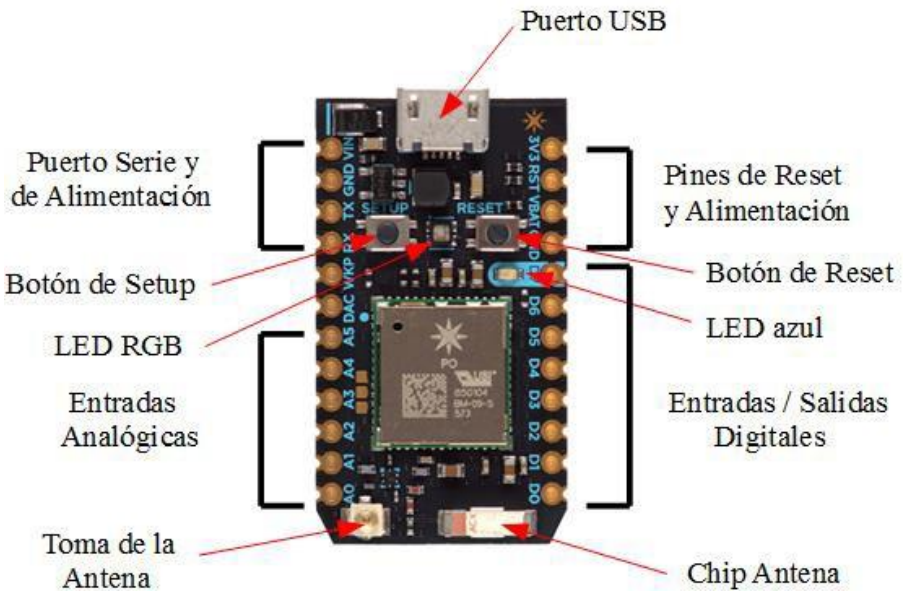

Figura 14. Componentes del dispositivo Photon [6]

Los botones principales *Setup* y *Reset* son los encargados de configurar nuevas credenciales Wi-Fi y reiniciar el dispositivo, donde la combinación de ambos permite el acceso a distintos modos de configuración internos del dispositivo *Photon*. Entre estos dos botones se encuentra un LED de tipo RGB *(Red, Green, Blue)* que proporciona información acerca del estado del dispositivo, por ejemplo, si se encuentra conectado a una red Wi-Fi estará parpadeando muy suavemente en color celeste o, si se está cargando un programa en *flash*, este LED parpadeará rápidamente en color rosa.

En el puerto micro-USB, localizado en la parte superior del dispositivo *Photon* (aunque principalmente se usa para alimentación), permite la comunicación serie, por lo que es posible la programación del hardware por esta vía. En la parte izquierda se encuentran los pines de puerto serie y de alimentación, compuestos por el pin VIN, GND, Transmisión (TX) y Recepción (RX) de comunicación serie. A la derecha se encuentran los pines de *reset* y alimentación (3V3, RST, VBAT y GND). El pin 3V3 se utiliza para alimentación externa, dado que toda la lógica del dispositivo funciona a esta tensión. La alimentación del dispositivo en sí es de 5V, siendo posteriormente convertida a 3.3V.

El pin RST tiene la misma función que el botón *Reset*: cuando el dispositivo *Photon* lee un nivel lógico alto en ese pin, reinicia el sistema. El pin VBAT permite la alimentación del dispositivo mediante una batería mientras éste se encuentra en modo *deep sleep*, con el fin de poder conservar así el contenido de su memoria. Existen ocho pines digitales de propósito general, D0 a D7, que pueden actuar como entradas o salidas. Profundizando un poco más en este aspecto, se demuestra que los pines D0 a D3 pueden usarse también como salidas analógicas utilizando técnicas *PWM (Pulse Width Modulation)*. Como se aprecia en la Figura 14, existe un LED integrado en el pin D7.

Los pines A0 a A5 constituyen entradas analógicas que trabajan con tensiones de entre 0 y 3.3V. Los pines analógicos también se pueden utilizar como entradas o salidas digitales, como los pines D0 a D7 y, al igual que los pines digitales, algunos pines analógicos (A4, A5), también se pueden utilizar como salidas analógicas PWM. A continuación, se encuentra el pin del conversor analógico-digital *(Digital Analog Converter, DAC)*, el cual se trata de un pin de salida analógica especial, capaz de proporcionar tensiones comprendidas entre 0 y 3.3V. A su lado se encuentra en el pin

WKP, utilizado para activar al dispositivo *Photon* después de que se ha puesto en modo *deep sleep*.

2.4.5 Dispositivo *RedBear Duo*

Para la implementación final de la pasarela se ha seleccionado el dispositivo *RedBear Duo*, mostrado en la Figura 15. Fabricado por la empresa RedBear [15], dispone de muchas similitudes con respecto al dispositivo *Photon* anteriormente analizado, que facilitarán la migración del *firmware* entre dispositivos.

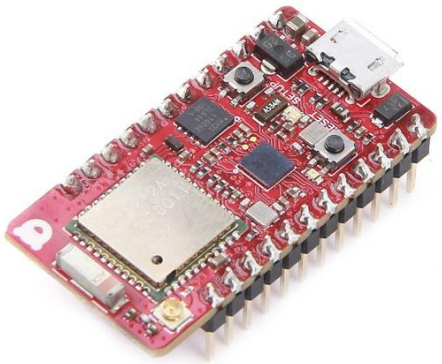

Figura 15. Dispositivo RedBear Duo

Este pequeño dispositivo de IoT incorpora un procesador STM32F205 ARM Cortex-M3 que, al igual que el dispositivo *Photon,* opera a una frecuencia de 120 MHz. Incluye 128 KB de SRAM y 1 MB de memoria *Flash* para almacenar el *firmware* del sistema.

Además de incluir el mismo chipset Wi-Fi BCM43438 de Broadcom que integra el dispositivo *Photon,* incluye un módulo Bluetooth 4.1 que permite proporciona conectividad BLE *(Bluetooth Low Energy)* especialmente útil para aplicaciones de bajo consumo.

La relación de aspecto y la disposición física de los pines de Entrada/Salida del dispositivo *RedBear Duo* es también similar a la del dispositivo Photon. En concreto, el dispositivo *RedBear Duo* cuenta con una gran cantidad de interfaces analógicas, digitales y de comunicación, tal y como se detalla en la Tabla 11.

Tabla 11. Periféricos del dispositivo RedBear Duo

Tipo de Periférico	Cantidad	Entrada(E)/Salida(S)
Digital	18	E/S
Analógico (ADC)	8	E
Analógico (DAC)	2	S
SPI	2	E/S
I2S	1	E/S
I2C	1	E/S
CAN	1	E/S
USB	1	E/S
PWM	13	S

En la Figura 16 se detallan los principales componentes del dispositivo *RedBear Duo*, mientras que en la Figura 17 se especifica su *pin-out*.

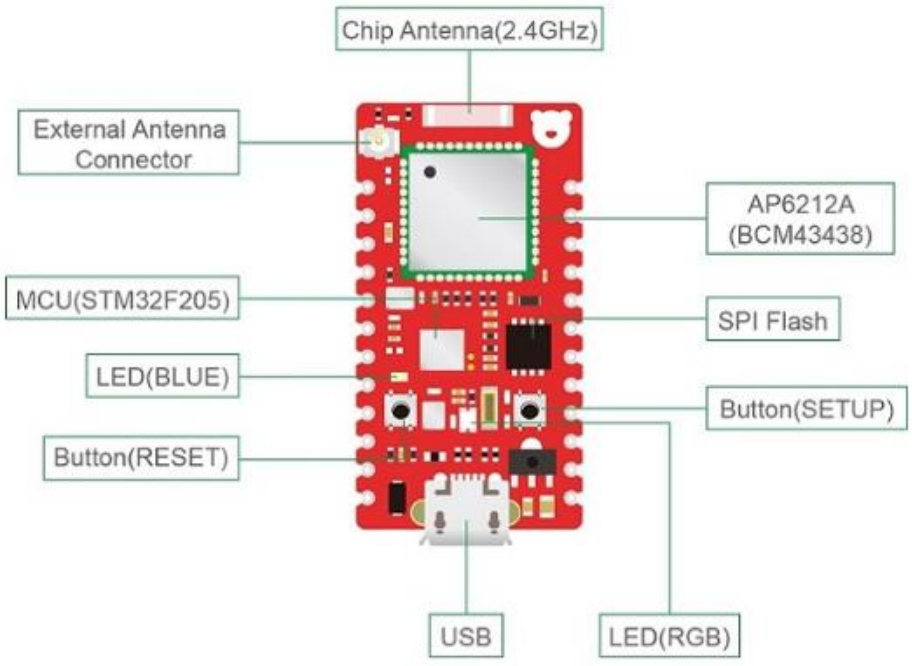

Figura 16.Componentes principales del dispositivo RedBear Duo

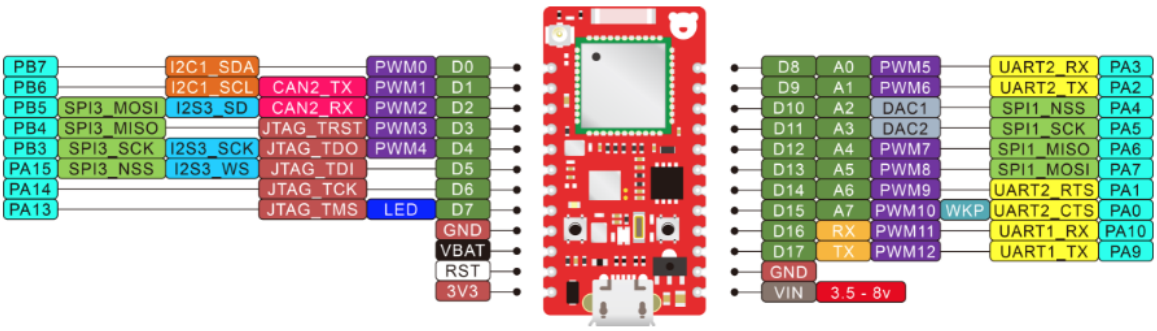

Figura 17. Pin-out del dispositivo *RedBear Duo*

Los dos botones disponibles en la plataforma (*Setup* y *Reset*) permiten configurar nuevas credenciales WiFi y reiniciar el dispositivo, entre otras funciones básicas. También se pueden utilizar en conjunto para provocar el restablecimiento de los valores de fábrica por defecto. Entre ambos botones se encuentra un LED de tipo RGB que proporciona información acerca del estado del dispositivo.

En la parte inferior se encuentra el puerto micro-USB, cuyo propósito principal es proporcionar alimentación al dispositivo *RedBear Duo*, aunque también se puede utilizar para la programación del dispositivo, así como para realizar comunicaciones serie USB con un ordenador. A la izquierda del puerto micro-USB se encuentran los pines de *reset* y alimentación (3V3, RST, VBAT y GND). El dispositivo *RedBear Duo* convierte la energía de entrada proporcionada a través de la alimentación del puerto micro-USB, o del pin VIN, en un suministro de 3.3V, ya que toda la lógica del dispositivo funciona con esta tensión. El pin RST se puede utilizar, al igual que el botón *Reset*, para reiniciar el sistema. El pin VBAT permite conectar una pequeña batería de reserva externa al dispositivo *RedBear Duo*, con el fin de retener el contenido de los registros RTC *(Real-Time Clock)*, hacer una copia de seguridad de la memoria SRAM y suministrar el RTC cuando se deje de suministrar la tensión VDD. Es importante destacar que la batería establecida en el pin VBAT no puede alimentar al dispositivo *RedBear Duo*.

Los pines D0 a D7 son pines de propósito general que pueden actuar como entradas o salidas digitales. Además, los pines D0 a D4 también pueden actuar como salidas analógicas utilizando técnicas PWM *(Pulse-Width Modulation)*. Por otro lado, tal y como se aprecia en la Figura 16, existe también un LED azul situado junto al pin D7 que se encuentra conectado directamente a este pin.

Este dispositivo integra el módulo AMPAK A6212A, construido en torno al circuito integrado BCM43438 de Broadcom, que soporta WiFi 802.11b/g/n y BLE 4.0, lo que permite desarrollar aplicaciones que se comuniquen con otros dispositivos inalámbricos a través de BLE, WiFi, o BLE+WiFi al mismo tiempo. Además, dispone también de un pequeño conector UFL a través del cual se puede conectar una antena externa. Con esto se conseguiría ampliar el rango de cobertura WiFi, añadiendo una antena más sensible o direccional. Por defecto, el dispositivo *RedBear Duo* intentará elegir la mejor antena, aunque también se puede controlar qué antena utilizar mediante *firmware*.

Los pines A0 a A6 constituyen entradas analógicas que operan con tensiones comprendidas entre 0V y 3.3V. Los pines analógicos también se pueden utilizar como entradas o salidas digitales, al igual que los pines D0 a D7 y, del mismo modo que los pines digitales, algunos pines analógicos (A0, A1, A4, A5 y A6) se pueden utilizar como salidas analógicas PWM. Asimismo, los pines A2 y A3 se pueden emplear como salidas de los conversores analógico-digitales *(Digital Analog Converter, DAC)* internos, los cuales constituyen dos pines analógicos de salida especiales capaces de proporcionar tensiones comprendidas entre 0V y 3.3V. A continuación del pin A6 se encuentra el pin WKP, el cual se utiliza para despertar al dispositivo *RedBear Duo* después de que se haya configurado en modo *deep sleep*, aunque también se puede emplear como salida analógica (A7), como entrada o salida digital (D15), y como salida analógica PWM.

Los pines TX y RX (transmisión y recepción) se utilizan para la comunicación serie. Por último, situados por debajo de estos pines se encuentran, un segundo pin GND, y el pin VIN. Tal y como se comentó anteriormente, se puede alimentar el dispositivo *RedBear Duo* suministrando entre 3.5V y 8V al pin VIN, como alternativa al uso del puerto USB.

El dispositivo *RedBear Duo* soporta varios tipos de lenguaje de programación, como Arduino, C/C++, JavaScript y Python, y las aplicaciones se pueden desarrollar con GCC, el entorno de desarrollo WICED SDK, o con los IDE de Arduino, Particle y Espruino.

En cuanto al tipo de aplicaciones, el dispositivo *RedBear Duo* se puede utilizar para el desarrollo de aplicaciones en diferentes ámbitos, como: Automatización industrial, automatización de edificios, electrodomésticos inteligentes, juguetes inteligentes, sensores IoT, WiFi/BLE *Gateway* o *Beacon Management*.

Además, debido a que dispositivo *RedBear Duo* es compatible con la ejecución de aplicaciones de usuario basadas en el *firmware* de la empresa Particle, en el proceso de fabricación del dispositivo se instala el *firmware* personalizado de Particle de forma predeterminada. En la Figura 17 se muestra exactamente la asignación de memoria en el dispositivo RedBear Duo, a partir de la versión v0.3.0 del *firmware* de usuario.

	Sector	Internal Flash Address	Size	Photon (for Particle)	Duo (for Particle/Arduino)	Duo (for WICED)
Internal Flash	0	0x08000000 - 0x08003FFF	16K	Bootloader	Bootloader (Modified)	Bootloader (Modified)
	1	0x08004000 - 0x08007FFF	16K	DCT1		
	2	0x08008000 - 0x0800BFFF	16K	DCT2	DCT1	DCT1
	3	0x0800C000 - 0x0800FFFF	16K	EEPROM 1	DCT2	DCT2
	4	0x08010000 - 0x0801FFFF	64K	EEPROM 2	Reserved	User-part
	5	0x08020000 - 0x0803FFFF	128K	System-part1	System-part1	
	6	0x08040000 - 0x0805FFFF	128K		System-part2	
	7	0x08060000 - 0x0807FFFF	128K	System-part2		
	8	0x08080000 - 0x0809FFFF	128K			
	9	0x080A0000 - 0x080BFFFF	128K	User-part		
	10	0x080C0000 - 0x080DFFFF	128K	OTA Image	User-part	
	11	0x080E0000 - 0x080FFFFF	128K	Factory Reset Image		
Exteral Serial Flash	0 - 183	0x00000000 - 0x000B7FFF	736K	No Ext. Flash	User Data	WICED
	184 - 187	0x000B8000 - 0x000BBFFF	16K		EEPROM 1	Reserved (Particle)
	188 - 191	0x000BC000 - 0x000BFFFF	16K		EEPROM 2	
	192 - 319	0x000C0000 - 0x0013FFFF	512K		OTA Images	
	320 - 383	0x00140000 - 0x0017FFFF	256K		Factory Reset Image	
	384 - 511	0x00180000 - 0x001FFFFF	512K		WiFi Firmware	

Figura 17. Mapa de memoria del dispositivo RedBear Duo

La aplicación de usuario se denomina *User-part*. La aplicación de usuario basada en el *firmware* de Particle se extiende desde la dirección de memoria *Flash* interna 0x080C0000, hasta la 0x080FFFFF, abarcando un tamaño de 256KB. Se compone de las funciones *setup()* y *loop(),* al igual que en el caso del dispositivo *Photon*.

En cuanto a las funciones que proporciona el firmware del dispositivo *RedBear Duo* relativas a conexiones inalámbricas BLE, incluye funciones propias de un dispositivo *Peripheral* y funciones propias de un dispositivo *Central*. Se proporcionan además funciones relativas a los roles *GATT Server* y *GATT Client*, definidos en BLE, que en el caso concreto del presente TFG son roles que adoptan el dispositivo *Peripheral* y el dispositivo *Central* (dispositivo móvil), respectivamente.

2.4.6 Módulo *Carloop*

Carloop es un kit de desarrollo de código libre que proporciona una interfaz de conexión entre el conector OBD-II de un vehículo, y un dispositivo IoT [16]. Esta interfaz, mostrada en la Figura 18, está especialmente diseñada inicialmente para acoplarse con la disposición de pines del dispositivo *Photon* de la empresa Particle*,* creando así una pasarela transparente entre la programación del *firmware* del dispositivo IoT y el Bus CAN del vehículo.

Figura 18. Interfaz Carloop para dispositivos IoT

Debido al carácter libre del kit de desarrollo, en el desarrollo del presente TFG ha sido posible utilizar las librerías que proporciona el fabricante para el uso de este dispositivo orientado a conectar los vehículos a la nube.

Antes de comenzar con el desarrollo del *firmware* del dispositivo, se hace patente la necesidad de comprobar la compatibilidad entre la interfaz *Carloop* y los dispositivos que se van a utilizar en este trabajo -*Photon y RedBear Duo-.*

Carloop viene definido por el fabricante como compatible con el dispositivo *Photon*, por lo que se realizará la comparativa de *pin-out* con el fin de evaluar su correcto funcionamiento.

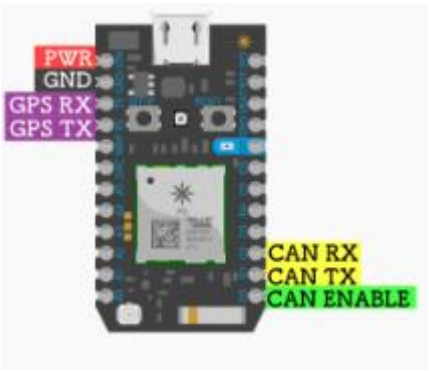

Figura 19. Pines utilizados por Carloop

La interfaz del módulo *Carloop* utiliza los pines que se destacan en la Figura 19, es decir, los pines relacionados con la alimentación del sistema (PWR y GND), emisor y receptor de información GPS (GPS TX/GPS RX), habilitador del Bus CAN (CAN ENABLE), y los pines relacionados con el envío y recepción de información por este bus (CAN TX y CAN RX).

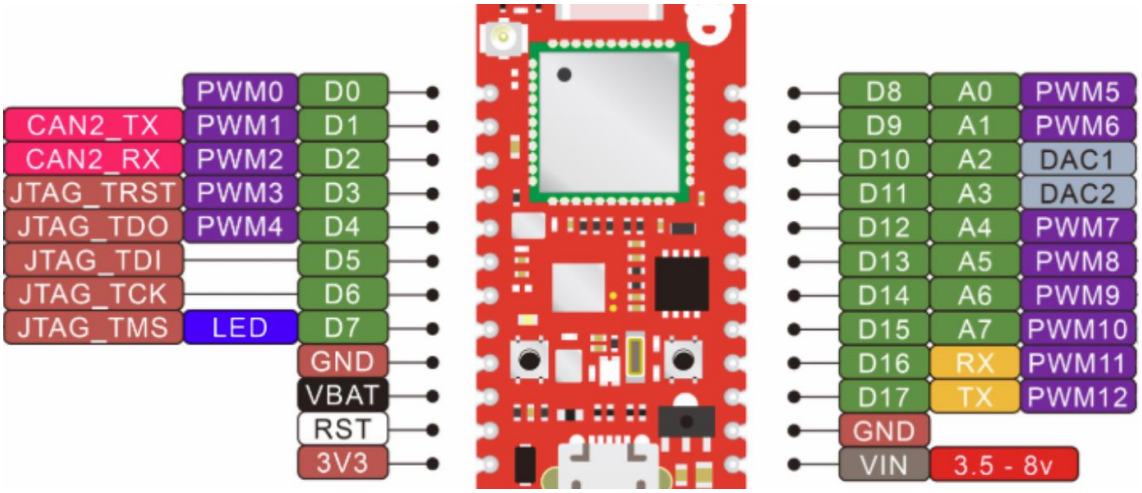

Figura 20. Pinout RedBear Duo

Como se aprecia en la Figura 20, la distribución de los pines del dispositivo *RedBear Duo* es compatible con la interfaz *Carloop*. Teniendo en cuenta la rotación de 90º de la Figura 20 con respecto a la Figura 19, Los pines dedicados a la energía del dispositivo (representado como VIN y GND), las líneas de recepción y transmisión de GPS (D16 y D17 respectivamente) así como las líneas de bus CAN (D1 y D2) y habilitador de este que utiliza *Carloop* (D0), están dispuestas de manera idéntica al dispositivo *Photon.*

2.5 Herramientas *software*

En esta sección se detallarán los recursos *software* utilizados en el desarrollo del *firmware* de los dispositivos, así como para su posterior depurado y validación funcional.

2.5.1 Particle Build

Particle Build se corresponde con un entorno integrado en una versión web que permite desarrollar el código para el *firmware* de los dispositivos de la empresa Particle directamente en el navegador web [17]. Una vez el usuario se haya registrado, ya puede comenzar a codificar en el entorno.

En la Figura 21 se muestra la interfaz de usuario de este entorno de desarrollo con un código de ejemplo que enciende y apaga un led.

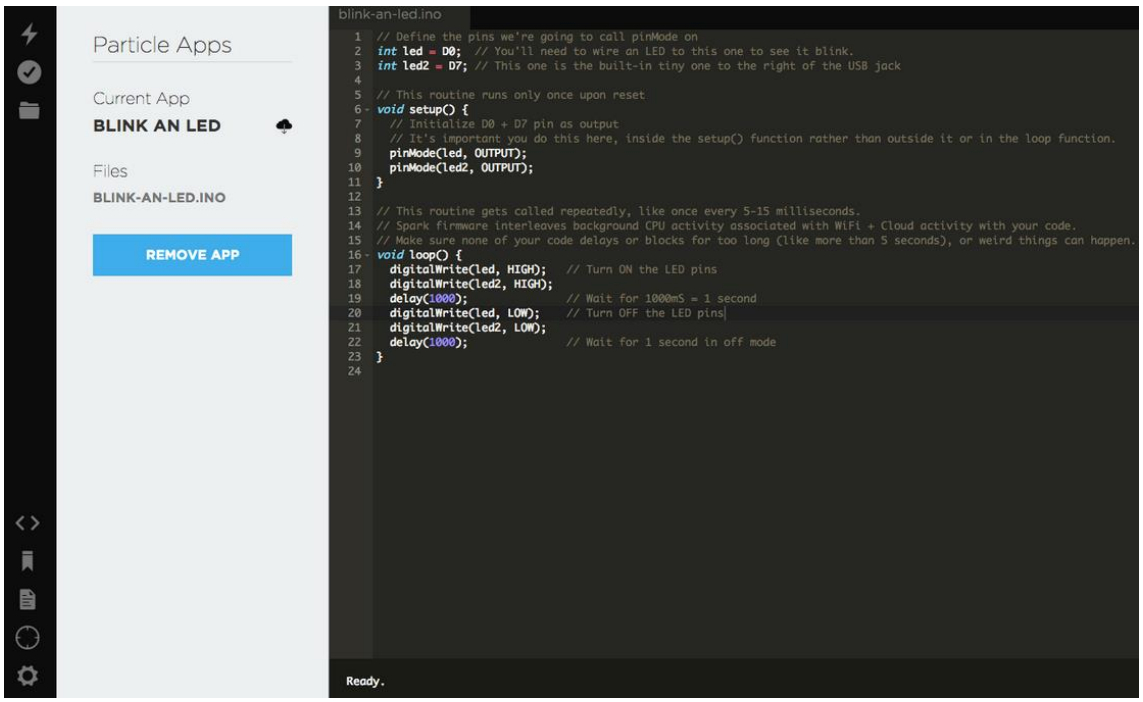

Figura 21. Entorno de desarrollo Particle

Este entorno se caracteriza por utilizar un diseño funcional, en el cual la barra vertical posicionada a la izquierda de la Figura 21 contiene los botones o accesos necesarios para el desarrollo de aplicaciones.

Comenzando desde arriba en la barra lateral izquierda, se encuentran tres iconos especialmente relevantes:

Flash: representado gráficamente con el símbolo de un rayo, permite cargar el *firmware* desarrollado en el dispositivo objetivo.

Verify: representado gráficamente con el símbolo de un tic de corrección rodeado de un círculo, permite al usuario compilar el código y detectar posibles errores en este.

Save: representado gráficamente con el símbolo de una carpeta, permite almacenar los cambios realizados en el código.

En la parte inferior de la barra lateral izquierda se encuentran cinco botones que son de especial utilidad para la navegación entre aplicaciones y el código. Estos son, por orden de aparición:

Code: Muestra una lista de las aplicaciones de usuario y permite navegar entre ellas.

Library: Permite navegar entre las diferentes librerías, tanto propias como creadas por los usuarios.

Docs: Permite el acceso a la documentación propia de *Particle.*

Devices: Mediante este botón se selecciona el dispositivo objetivo en el cual cargar el código desarrollado.

Settings: Apartado de configuración del entorno de desarrollo. Permite cambiar la contraseña, cerrar la sesión y obtener el *token* de acceso que contiene las credenciales de identificación del usuario que utiliza el entorno IDE de Particle.

Asimismo, en la sección ***Particle Apps*** se muestra el nombre de la aplicación actual, así como una lista con el resto de aplicaciones. La aplicación abierta en el editor se muestra con el encabezado *Current App*, y debajo se detallan los archivos y las librerías incluidas en la aplicación. En este panel se encuentran los siguientes botones que permiten administrar la biblioteca de aplicaciones: crear, eliminar, cambiar el nombre, e incluso

ejemplos de aplicaciones que sirven de referencia para crear aplicaciones propias, como se muestra en la Figura 22.

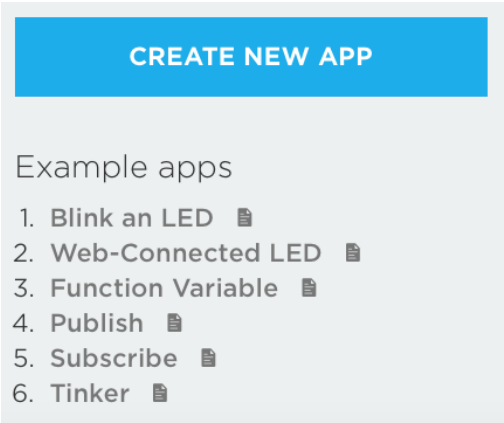

Figura 22 .IDE Particle Build: Ejemplos de aplicaciones

Por último, en la parte derecha del entorno de desarrollo se encuentra localizado el panel en el que se escribirá el código necesario para la aplicación de usuario que se desee implementar. En la parte inferior de dicho panel se halla una sección que determina el estado del código, mostrando los errores, el tamaño de memoria utilizada en el dispositivo, o si se ha realizado la programación del dispositivo de forma satisfactoria, entre otros.

Todos los procedimientos se realizan en la nube, desde el proceso de compilación del código hasta la programación del dispositivo.

Cabe destacar que la estructura que siguen todos los programas desarrollados para el dispositivo *Photon* (formato `.ino`) consta de dos funciones básicas: *setup()* y *loop()*. La función *setup()* es utilizada normalmente para inicializar pines y objetos, y sólo se ejecuta una vez al cargar el código en la memoria *Flash* del dispositivo, mientras que dentro de la función *loop()* se desarrolla el código que define el funcionamiento de la aplicación de usuario.

2.5.2 Interfaz de línea de comandos (CLI) de *Particle*

Además de *Particle Build*, la empresa Particle ofrece la herramienta *Particle CLI (Command Line Interface)* [17], que permite interactuar con la nube de Particle y sus dispositivos utilizando el lenguaje de programación *Node.JS* a través de una serie de

comandos definidos que pueden ejecutarse desde la consola o terminal del sistema, tanto en Windows como en Mac OS X y Linux.

2.5.3 Librería *Carloop*

Dentro del kit de desarrollo de *Carloop* se incluye una librería para su utilización. Estas librerías simplifican el uso de la interfaz *Carloop* al incluir funciones básicas que proporciona un nivel de abstracción mayor al código [18].

Entre las funciones incluidas en la librería, las más relevantes para el desarrollo de este TFG son las relacionadas con la utilización del bus CAN, para lo cual se creará un objeto *Carloop.*

En apartados posteriores se analizarán en profundidad las funciones de la librería utilizadas para el desarrollo del *firmware* de los dispositivos.

2.5.5 PuTTY

Con el objetivo de transmitir los comandos vía puerto serie desde el ordenador portátil hasta el dispositivo *Photon,* se ha utilizado el programa de código libre *PuTTY*, cuyo icono se muestra en la Figura 23 [19]. Este *software* es un cliente *SSH*, *Telnet*, *rlogin* y *TPC Raw* programado en C, que permite configurar la conexión serie del dispositivo e incluye una ventana para la visualización de datos.

Figura 23. Icono del software PuTTy

2.5.6 nRF Connect para Android

La aplicación *nRF connect*, cuyo icono se muestra en la Figura 24 [20] es el nombre que recibe el *software* que se ha utilizado para la comprobación y correcto desarrollo de la plataforma final HW/SW. Esta aplicación móvil permite utilizar la tecnología *Bluetooth Low Energy* que integra el teléfono móvil utilizado para el desarrollo de este TFG.

Figura 24. Icono aplicación nRF Connect

Gracias a este *software* ha sido posible realizar el análisis en profundidad de las tramas de información BLE, a la vez que ha permitido la interacción con el dispositivo IoT *RedBear Duo*, actuando como dispositivo *Central.*

3 Implementación de la interrogación vía serie

Con objetivo de comprobar el correcto funcionamiento de las comunicaciones entre la centralita del vehículo y el módulo formado por la interfaz de *Carloop* y el dispositivo *Photon*, se ha realizado una comprobación con un firmware de prueba que además sirve como punto de partida para la posterior codificación del *firmware* de interrogación de PID.

3.1 Objetivos de la implementación

Como se analizó en el capítulo 2.3.6 “Módulo Carloop”, los dispositivos *Photon* y *RedBear Duo* son compatibles con la interfaz de conexión *Carloop*. Aprovechando esta interoperabilidad se ha podido codificar el *firmware* encargado de realizar las peticiones y recepciones de los PID (interrogación) en el bus de comunicaciones del vehículo.

Como primer paso de cara a la implementación de la plataforma HW/SW final basada en el dispositivo *RedBear Duo,* se llevó a cabo la implementación de su funcionalidad básica con prestaciones reducidas, utilizando el dispositivo *Photon* montado en el módulo *Carloop,* como se muestra en la Figura 25. Con esto se pretende comprobar y verificar la funcionalidad del *firmware* de interrogación y solventar las posibles dificultades que presente.

Las prestaciones reducidas se basan en la posibilidad de realizar peticiones de manera secuencial con direcciones de PID únicas. Se ha escogido solo el primer *bitmap* de los tres analizados en el capítulo 2.3.3 “Direcciones de PID estándar” con el fin de reducir el número de parámetros de decodificación necesarios para el análisis de la respuesta proporcionada por la centralita del vehículo.

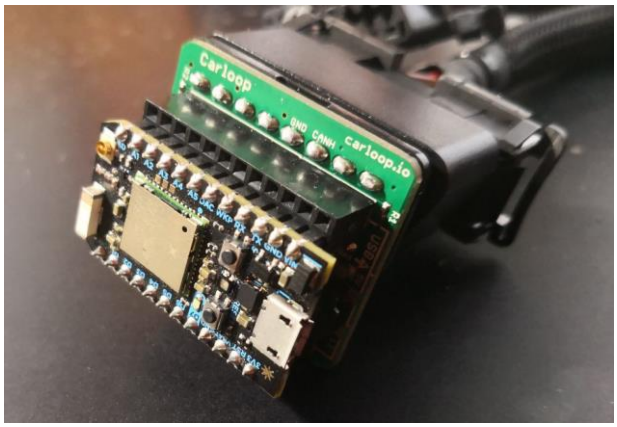

Figura 25. Dispositivo Photon montado sobre la interfaz Carloop

Las peticiones son realizadas por medio de la interfaz serie disponible en el dispositivo *Photon* que, desde un portátil que envía los comandos, es el encargado final de la transmisión y recepción de la información disponible en el bus de comunicaciones del vehículo, y de su transmisión al portátil.

3.2 Librería Carloop

Con objetivo de codificar el *firmware* de los dispositivos *Photon* y *RedBear Duo*, se ha utilizado la librería proporcionada por el kit de desarrollo *Carloop* [21]. En ella, se especifican las funciones implementadas para la comunicación mediante tramas con el bus CAN del vehículo. Entre ellas, las utilizadas para la realización del *firmware* de los dispositivos son dos:

- *Carloop.can().receive(CAMMessage object).*
- *Carloop.transmit (CANMessage object).*

La función definida como "*Carloop.can().receive(CAMMessage object)"* será la encargada de procesar la recepción de las tramas CAN. Esto será posible por medio del uso del objeto *"CANMessage"* incluido en la documentación de *Particle*, que será analizado más adelante.

De manera análoga, la función "*Carloop.transmit (CANMessage object)"* permitirá el envío de una trama CAN por medio de otro objeto del tipo *"CANMessage"*.

En el fragmento de código mostrado en la Figura 26 se indican las variables que se incluyen en este objeto. En este trabajo se hará hincapié en los campos conformados por

la ID del mensaje de CAN (*"id"* en el código), la longitud del mensaje (*"len"* en el código) y el vector de bytes de información, que serán los encargados de almacenar la información a transmitir en el bus de comunicaciones (*"data[8]"* en el código).

```
1. struct CANMessage
2. {
3.    uint32_t id;
4.    bool     extended;
5.    bool     rtr;
6.    uint8_t  len;
7.    uint8_t  data[8];
8. }
```

Figura 26. Estructura CANMessage

Después de generar la estructura compatible con las funciones de *carloop,* es posible realizar transmisiones mediante la función "*Carloop.transmit (CANMessage object)"* utilizando como argumento el objeto *CANMessage* con los campos configurados específicamente para el propósito de este trabajo.

De manera análoga, la función "*Carloop.can().receive(CAMMessage object)* realizará el proceso de recepción de una trama CAN en la cual se identifican los mismos campos que se han analizado en el fragmento anterior.

3.3 Diagrama de flujo

En la Figura 27 se aprecia un diagrama de flujo del *firmware* que se pretende implementar. En él, el usuario, ejemplificado con el portátil que envía los comandos y recibe la información, utilizará la interfaz serie para realizar las llamadas a las funciones implementadas.

En caso de seleccionarse la opción 1 en la interfaz serie, se ejecutará la rutina encargada de descubrir los PID, que procederá a enviarla de nuevo vía serie mostrando un *bitmap* de 32 *bits*, en el cual los "1" representarán direcciones PID disponibles.

Si por otro lado se ha seleccionado la opción 2, el sistema pedirá al usuario vía serie que introduzca un *bitmap* de 32 *bits* en el cual introduzca un valor "1" en la posición de la dirección del PID que desee recibir información vía serie.

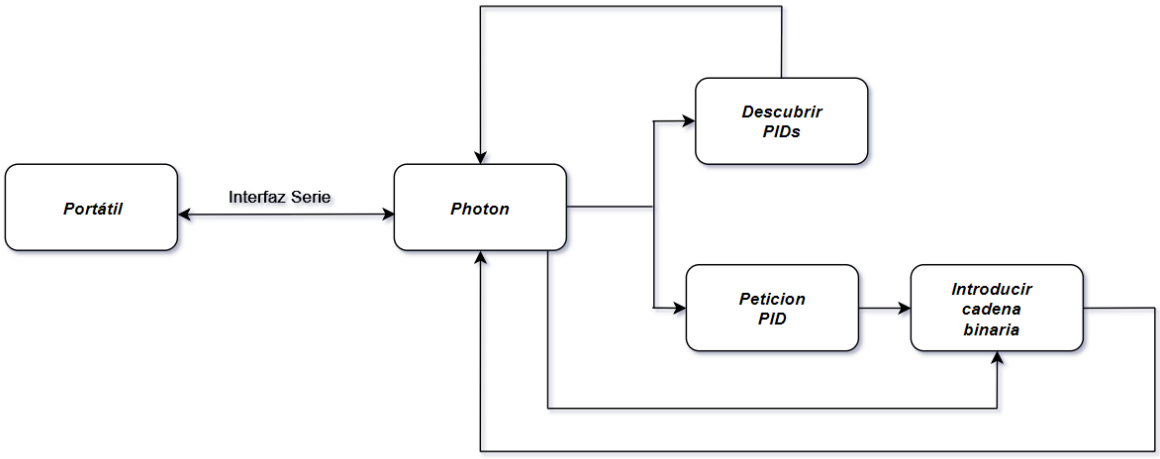

Figura 27. Diagrama de flujo firmware Photon

3.4 Desarrollo del *firmware*

3.4.1 Punto de partida y verificación del bus de comunicaciones

En esta sección del TFG se abordará la elaboración del código que se ha utilizado para comprobar el correcto funcionamiento del bus de comunicaciones, a partir de la transmisión y recepción de información entre los diferentes módulos que constituyen la plataforma inicial realizada a partir del dispositivo *Photon*.

En el fragmento mostrado en la Figura 28 se definen los parámetros iniciales del código, entre los que se incluyen la librería *Carloop* (creando un objeto *carloop*) y la declaración de las funciones que se van a utilizar (*enviarPeticion()*, *esperarRespuesta()* y *margenPeticion*), así como las diferentes variables y constantes utilizadas para el correcto funcionamiento del código.

```
1. //Funciones
2. void enviarPeticion();
3. void esperarRespuesta();
4. void margenPeticion();
5.
6. Carloop<CarloopRevision2> carloop;
7.
8. // IDs de los mensajes OBD
9. const auto ID_peticion      = 0x7E0;
10.      const auto OBD_REPLY_ID        = 0x7E8;
11.      const auto ID_broadcast    = 0x7DF;
12.      const auto OBD_PID_SERVICE     = 0x01;
13.      const auto ID_respuesta_minima = 0x7E8;
14.      const auto ID_respuesta_maxima = 0x7EF;
15.
```

```
16.
17.      // Constantes para guardar las direcciones de los pids.
18.      // const auto AVALIBLE_PIDS = 0x00;
19.
20.      const auto VEHICLE_SPEED          = 0x0D;
21.
22.
23.      // asignacion de dirección de un pid.
24.
25.      const uint8_t pid = VEHICLE_SPEED;
26.
27.
28.      // funcion automatica para continuar mandando request
29.      auto *obdLoopFunction = enviarPeticion;
30.
31.      //generamos la variable que nos ayudara a controlar el numero
   de peticiones
32.      unsigned long transitionTime = 0;
```

Figura 28. Definiciones del firmware de verificación para Photon

Las ID definidas para las peticiones son *ID_peticion* y *ID_respuesta*, las cuales funcionan como se ha definido en el apartado 2.2.2 “Formato de petición y respuesta”. En caso de realizar la petición con la dirección definida por *ID_peticion* (0x7E0), se espera obtener una respuesta con la dirección 0x7E8 definida como *ID_respuesta*.

Para aportar más seguridad a la prueba, se ha añadido la dirección *ID_broadcast*, que posibilita analizar un rango de direcciones de respuesta más amplio. Estos rangos están determinados por el valor de las constantes *ID_respuesta_minima* e *ID_respuesta_maxima*.

Se ha incluido también la dirección definida por *OBD_PID_SERVICE*, que hace referencia al modo de operación definido por el estándar SAE, y analizado en el capítulo 2.2.1 “Modos de operación”.

Como última declaración, se ha definido la función de *looping* automático utilizada para la correcta secuencia de llamadas a las funciones utilizadas para el envío y recepción de datos, así como la dirección del PID que se va a utilizar en la trama de petición (*VEHICLE_SPEED*).

Una vez definidas las constantes y las variables necesarias, se procede a codificar la función *setup()* de inicialización principal de la aplicación. Esta codificación se encuentra en el fragmento de código mostrado en la Figura 29.

```
1. //configuramos los parametros de inicializacion
```

```
2. void setup()
3. {
4.     Serial.begin(9600);
5.     carloop.begin();
6.     transitionTime = millis();
7. }
```

Figura 29. Función setup()

Se configura el puerto serie a una velocidad de 9600 baudios, se inicia el objeto *carloop*, y se genera una variable de tiempo utilizada para la secuencia de funciones.

Por su parte, en el fragmento de código representado en la Figura 30 se codifica la función *loop()* principal del sistema, que se basa en la ejecución de la función de actualización del objeto *carloop* y en la ejecución de la función que secuencia la recepción y envío de tramas.

```
1. //loop del programa
2. void loop()
3. {
4.     carloop.update();
5.     obdLoopFunction();
6. }
```

Figura 30. Función loop()

Para finalizar, se indica a continuación la codificación del resto de las funciones implementadas en para la implementación de la funcionalidad de esta prueba.

EnviarPeticion() - Envío de trama de Bus CAN

Como se observa en el fragmento de código representado en la Figura 31, esta función se encarga de crear un objeto mensaje de tipo *CANMessage* que incluirá los parámetros necesarios para el envío de tramas reconocibles por parte de la centralita del vehículo. Para la prueba se ha configurado para realizar una petición en el único PID declarado (*VEHICLE_SPEED*).

```
1. //enviar una peticion CAN
2. void enviarPeticion() {
3.
4.     CANMessage mensaje;
5.     mensaje.id = OBD_BROADCAST_ID; //Id del tipo de mensaje
6.     mensaje.len = 8; //longitud del mensaje (caracteristica del
   objeto message)
7.     mensaje.data[0] = 0x02; //numero de datos adicionales
8.     mensaje.data[1] = OBD_PID_SERVICE; //modo de operacion
```

```
9.      mensaje.data[2] = pid; //direccion pid objetivo
10.           carloop.can().transmit(mensaje);
11.
12.           obdLoopFunction = esperarRespuesta;
13.           transitionTime = millis();
14.       }
```

Figura 31. Función EnviarPetición()

El envío de la trama se realiza por medio de la función *carloop.can().transmit(mensaje)* con el formato de petición definido, configurando el campo ID con la dirección *broadcast* (que presumiblemente aportará más robustez al código) y ejecutando una función secuencial para establecer un tiempo de respuesta al vehículo.

esperarRespuesta() - Recepción de trama de Bus CAN

Una vez que se ha ejecutado la función *"enviarPeticion()"* y se haya esperado un tiempo suficiente para el correcto funcionamiento del bus, se procede a recibir la respuesta del vehículo a partir del fragmento de código mostrado en la Figura 32.

```
1. void esperarRespuesta() {
2.
3.     if(millis() - transitionTime >= 100) {
4.         obdLoopFunction = margenPeticion;
5.         transitionTime = millis();
6.         return;
7.     }
8.
9.     bool respuesta = false;
10.
11.           CANMessage mensaje;
12.           while(carloop.can().receive(mensaje)) {
13.               Serial.print("mensaje.id = ");
14.               Serial.println(mensaje.id, HEX);
15.               Serial.print("mensaje.data[2] = ");
16.               Serial.println(mensaje.data[2], HEX);
17.
    if(mensaje.id >= ID_respuesta_minima && mensaje.id <= ID_respuest
  a_maxima && mensaje.data[2] == pid) {
18.                   respuesta = true;
19.                   Serial.println("respuesta OK");
20.                   return;
21.               }
22.           return;
23.           }
24.       }
```

Figura 32. Función esperarRespuesta()

La función *"enviarPeticion()"* genera el objeto *CANMessage "mensaje"*, que analizará las tramas que recibe desde el bus de comunicaciones hasta que la ID de una trama corresponda con una ID incluida en el rango de respuestas esperadas.

En el fragmento de código representado en la Figura 30 se indica el envío del mensaje *"respuesta OK"* vía serie en caso de que una de las respuestas recibidas estuviera dentro del rango de direcciones que existe entre *"ID_respuesta_minima"* e *"ID_respuesta_maxima",* con el filtro de direcciones ID. Con objetivo de realizar las tareas de depuración y corrección de manera efectiva, también se añadieron mensajes vía puerto serie para mostrar por pantalla el ID y el segundo *byte* (dirección PID) de la información recibida en cada trama CAN.

3.4.2 Disponibilidad de PIDs y peticiones de información

Una vez comprobado el correcto funcionamiento de la comunicación básica entre el dispositivo IoT *Photon* y el bus de comunicaciones del vehículo, se procedió a desarrollar el *firmware* con prestaciones limitadas.

Estas prestaciones posibilitan al dispositivo IoT realizar la consulta de PID disponibles al vehículo, conociendo así las direcciones disponibles de información de la centralita del vehículo, así como realizar peticiones secuenciales de PID para recibir la información relativa a estos.

En este caso se ha partido del *firmware* analizado en el capítulo anterior 3.2.1 "Punto de partida y verificación del bus de comunicaciones", al cual se le han añadido las funciones asociadas a la implementación de las prestaciones reducidas que se han establecido, así como una interfaz serie para la correcta comunicación entre el portátil y el dispositivo IoT.

```
1. #define NUMBER_AVAILABLE_PIDS 32
2.
3. //declaracion de funciones
4.
5. String transformarcadena();
6. void DescubrirPIDS();
7. void PeticionPID();
8. bool respuesta = false;
```

Figura 33. Definiciones del firmware del dispositivo Photon

Como se puede observar en el fragmento de código mostrado en la Figura 33, en primer lugar, se ha definido una constante denominada *NUMBER_AVAILABLE_PIDS*, con un valor de 32 que se corresponde con el número máximo de bits incluidos en los 4 *bytes* de información que puede contener una trama de CAN, de acuerdo con los formatos de petición y respuesta. Se han modificado las funciones del código eliminando la función que establecía el margen de tiempo entre petición y respuesta. Este cambio será analizado en la codificación de las funciones *"DescubrirPIDS()"* y *"PeticionPID()"*.

Asimismo, se ha incluido una función encargada de la transformación de variables de tipo *float* en cadenas binarias. Esta función, denominada *"transformarcadena()"* será de especial utilidad para implementar en el dispositivo *Photon* la funcionalidad de comprobar qué PID tiene disponible el vehículo para su consulta, y transmitir esta información por el puerto serie.

En la línea 8 del fragmento de código mostrado en la Figura 34 se encuentra la declaración de una variable de tipo *booleana*, denominada *"respuesta"* que se utiliza como medio para arbitrar las peticiones y respuestas por parte del dispositivo *Photon*, dentro del bus de comunicación.

```
1.
2. //direcciones cadenas de PIDs
3. const auto Disp_PIDS_01_20 = 0x00;
4. const auto Disp_PIDS_21_40 = 0x20;
5. const auto Disp_PIDS_41_60 = 0x40;
6. const auto Disp_PIDS_61_80 = 0x60;
7. //Direcciones PIDs
8.
9. // IDs de los mensajes OBD
10.
11.      const auto ID_respuesta       = 0x7E8;
12.      const auto ID_peticion      = 0x7E0;
13.      const auto ID_broadcast      = 0x7DF;
14.      const auto ID_respuesta_minima = 0x7E8;
15.      const auto ID_respuesta_maxima = 0x7EF;
16.      const auto OBD_PID_SERVICE     = 0x01;
17.
18.      const size_t numero_pids = 4;
19.
20.      const uint8_t pidsToRequest[numero_pids] = {
21.      Disp_PIDS_01_20,
22.      Disp_PIDS_21_40,
23.      Disp_PIDS_41_60,
24.      Disp_PIDS_61_80
25.      };
```

Figura 34. Definiciones firmware Photon

En el fragmento de código mostrado en la Figura 34 se recogen las declaraciones relacionadas con las direcciones de los cuatro PID encargados de identificar los PID disponibles en el vehículo. Estas direcciones componen los cuatro conjuntos de 32 bits que dispone el vehículo para señalizar qué PID tiene disponibles.

Una vez definidas las direcciones para consultar la disponibilidad de los PID, se procede a utilizar la codificación ya conocida de las direcciones correspondientes a los ID que se utilizan para la correcta trasmisión de información.

Para una mayor practicidad, se ha incluido una constante que agrupe el conjunto de direcciones de los PID de disponibilidad.

```
1. //variables
2. float dato0;
3. float dato1;
4. float dato2;
5. float dato3;
6. float dato4;
7. float dato5;
8. float dato6;
9. float dato7;
10.       String cadena_bin;
11.       String bin32;
12.       char inOption = {0};
13.       int in_bit;
14.       char in_bitmap[NUMBER_AVAILABLE_PIDS] = {0};
```

Figura 35. Definiciones firmware Photon

Para terminar con las declaraciones del *firmware* desarrollado, se han creado ocho variables del tipo *float* que almacenarán la información proveniente de los 8 *bytes* de información que devuelve el vehículo como respuesta ante cualquier petición, como se puede observar en el fragmento de código de la Figura 35. Las otras variables declaradas serán analizadas en profundidad en la sección de este apartado correspondiente a su uso dentro de las diferentes funciones implementadas.

```
1. void setup(){ carloop.begin();
2.   Serial.begin(9600);
3.   Serial.println("MENU:");
4.   Serial.println("-----------------------------------------------
   -");
5.   Serial.println("1. Descubrir PID");
6.   Serial.println("2. Introduzca PID para recibir informacion");
7.   Serial.println("-----------------------------------------------
   -");
8.   Serial.print ("Introduzca el numero de la accion que desee
   ejecutar: ");
```

```
9.
10.       }
```

Figura 36. Función setup()

Una vez realizadas las declaraciones ya descritas, se procede a realizar la codificación de la función *"setup()"* del *firmware*, mostrada en la Figura 36. Se establece la inicialización del puerto serie a la misma velocidad de 9600 baudios, y se crea una interfaz básica serie que facilite las comunicaciones entre el portátil y el dispositivo IoT.

Esta interfaz serie genera un menú contextual inicial que permite realizar las dos funciones básicas implementadas en este *firmware*. Así, en caso de que el usuario pulse la tecla "1", el sistema realizará automáticamente la petición de información al PID con dirección 0x00, que presumiblemente obtendrá como respuesta 8 *bytes* en los que se incluye la información de la cadena binaria de disponibilidad de los primeros 32 PID presentes en el vehículo.

Si por el contrario el usuario pulsa la tecla "2", el sistema le solicita que ingrese una cadena binaria de 32 bits con el objetivo de realizar una petición a una dirección PID específica.

El funcionamiento de las peticiones se analizará en profundidad en la codificación de las funciones *"PeticionPID()"* y *"DescubrirPIDS()"*.

Con el objetivo de realizar un análisis práctico de la función *"loop()"* correspondiente al *firmware* cargado en el dispositivo *Photon,* se ha fragmentado este módulo en tres partes diferenciadas. Por orden de aparición, los fragmentos corresponden a las posibles opciones que puede elegir el usuario.

En caso de que el usuario seleccione a opción "1", el fragmento de código correspondiente a la sección del código que procesa esta información es la mostrada en la Figura 37.

```
1. void loop(){
2. //comprobacion de la existencia de un caracter en el puerto serie
3.   if (Serial.available() > 0)  {
4.      char inOption = Serial.read();
5.
6.      Serial.println(inOption);
7.      if (inOption == '1') {
8.          Serial.println("opcion 1");
9.
```

```
10.             while(respuesta == false) {
11.                 DescubrirPIDS();
12.             }
13.             String pids_disp = bin32;
14.             Serial.println(pids_disp);
15.             respuesta = false;
16.
17.
18.              Serial.println();
19.              Serial.println("MENU:");
20.              Serial.println("-------------------------------------
   -------------");
21.              Serial.println("1. Descubrir PID");
22.              Serial.println("2. Introduzca PID para recibir
   informacion");
23.              Serial.println("-------------------------------------
   -------------");
24.              Serial.print ("Introduzca el numero de la accion que
   desee ejecutar: ");
25.           }
```

Figura 37. Función loop() opción 1

Como primera instancia de la función *loop()*, se comprueba la existencia de un carácter disponible en el puerto serie. Una vez que se detecta dicho carácter, y se comprueba que es un "1", el sistema ejecuta la función *"DescubrirPIDS()"*, que refresca el valor de la variable *"bin32"*. Una vez realizado este refresco se asigna el valor nuevo a una variable auxiliar denominada *"pids_disp"*, que será la que finalmente se muestre por pantalla.

Realizado esto, se asigna el valor *false* a la variable *booleana "respuesta"*, que permite realizar futuras peticiones en el menú contextual que se vuelve a generar al final del fragmento de código.

En caso de que el usuario seleccione la opción "2", la sección de la función *loop()* que lo procesa se corresponde con el fragmento de código representado en la Figura 38.

```
1. else if (inOption == '2') {
2.         // Recibir el bitmap de los grupos de PID a representar
3.
4.         in_bit = 0;
5.         Serial.print("Enter bitmap (32bits): ");
6.         while (in_bit < (NUMBER_AVAILABLE_PIDS)) {
7.            while (!(Serial.available() > 0)) Particle.process(); //
   se genera un bitmap de 32 bits con la informacion introducida por
   el usuario
8.            char inID = Serial.read();
9.            Serial.print(inID);
10.                in_bitmap[in_bit] = inID;
```

```
11.                 in_bit++;
12.             }
13.             Serial.println();
14.             int p = 0;
15.             while(in_bitmap[p]=='0') {
16.                  p++;
17.             }
18.             int k =p+1;
19.             respuesta = false;
20.             while (respuesta == false){
21.                 PeticionPID(k);
22.             }
23.             Serial.println();
24.             Serial.println("MENU:");
25.             Serial.println("-------------------------------------
   -------------");
26.             Serial.println("1. Descubrir PID");
27.             Serial.println("2. Introduzca PID para recibir
   informacion");
28.             Serial.println("-------------------------------------
   -------------");
29.             Serial.print ("Introduzca el numero de la accion que
   desee ejecutar: ");
30.           }
```

Figura 38. Función loop opción 2

Después de comprobar la disponibilidad de un carácter en el puerto serie, el sistema irá rellenando el *array "in_bitmap[in_bit]"* de 32 *bits* con los caracteres que el usuario introduzca. Dentro de las limitaciones de este *firmware* se encuentra la imposibilidad de realizar la consulta de más de una dirección PID por petición, por lo que solo se realizará la petición del primer "1" encontrado en la cadena binaria introducida por el usuario.

Con la información de la cadena binaria almacenada, el sistema comprueba, carácter a carácter, donde se encuentra el primer "1" de la cadena. El valor del índice en el cual se encuentra este valor se almacena en la variable *"p"*. Esta variable *"p"* se incrementa por medio de la variable *"k"*, con el fin de obtener la dirección PID en cuestión [Tabla 12].

Como se analizó en el apartado 2.2.3 "Direcciones de PID estandarizados", realizar una petición a la dirección 0x00 hará que el vehículo devuelva una cadena binaria de 32 *bits,* denotando con un "1" los que corresponden a posiciones de los PID disponibles, sin contar consigo mismo, es decir, el sistema devuelve la disponibilidad de los siguientes 32 PID para consulta.

Tabla 12. Ejemplo correspondencia bitmap introducido con la petición a realizar

Índice	0	1	2	3
Valor	0	1	0	0
Direccion PID	1	2	3	4

Si se realiza la petición de disponibilidad y se recibe, por ejemplo, un *array* de cinco posiciones [Tabla 12], La posición del valor "1" que denota la disponibilidad, se encuentra en el índice 1 del *array*, si bien corresponde con la dirección "2" (0x02) dentro del mapa de disponibilidad.

En el fragmento de código mostrado en la Figura 39 se aprecia el uso de esta información para generar el valor de la variable *"k"* que almacena la información del PID objetivo de la consulta. Una vez almacenada la información en la variable, se procede a ejecutar la función *"PeticionPID()"* con el argumento *"k"* en bucle condicionado por la variable *booleana "respuesta".*

El último fragmento correspondiente a la función *"loop()"*, mostrado en la Figura 39, corresponde a una protección en caso de que el usuario no pulse ninguna de las teclas definidas.

```
1. else {
2.          Serial.println("Not a valid option");
3.          Serial.println("");
4.
5.          Serial.println("MENU:");
6.          Serial.println("----------------------------------------
   -------");
7.          Serial.println("1. Descubrir PID");
8.          Serial.println("2. Introduzca PID para recibir
   informacion");
9.          Serial.println("----------------------------------------
   -------");
10.              Serial.print ("Introduzca el numero de la accion que
   desee ejecutar: ");
11.             }
12.         }
13.
14.      }
```

Figura 39. Función loop() opción no existente

En ese caso se le envía un mensaje al usuario que confirma que esa opción no es válida, y se le vuelve a mostrar el menú contextual.

DescubrirPIDS() - Disponibilidad de PIDs

La función *"DescubrirPIDS()"* se ejecuta en el código una vez que el usuario ha seleccionado la opción "1", y permitirá conocer qué direcciones PID están disponibles para su consulta en el vehículo.

En el fragmento de código mostrado en la Figura 40 se recoge la codificación de dicha función.

```
1. void DescubrirPIDS() {
2.     bin32 ="";
3.     CANMessage mensaje;
4.     mensaje.id = ID_peticion ; //Id del tipo de mensaje
5.     mensaje.len = 8; //longitud del mensaje (caracteristica del
   objeto message)
6.     mensaje.data[0] = 0x02;
7.     mensaje.data[1] = OBD_PID_SERVICE;
8.     mensaje.data[2] = pidsToRequest[0];
9.     carloop.can().transmit(mensaje);
10.          delay(600);
11.          respuesta = false;
12.
13.
14.          CANMessage mensaje2;
15.          while(carloop.can().receive(mensaje2)) {
16.
     if(mensaje2.id == ID_respuesta && mensaje2.data[2] == pidsToReque
   st[0]) {
17.
18.                   respuesta = true;
19.                   dato0 = mensaje2.data[0];
20.                   dato1 = mensaje2.data[1];
21.                   //PID code
22.                   dato2 = mensaje2.data[2];
23.                   //datos
24.                   dato3 = mensaje2.data[3];
25.                   dato4 = mensaje2.data[4];
26.                   dato5 = mensaje2.data[5];
27.                   dato6 = mensaje2.data[6];
28.                   //no tiene informacion de interes
29.                   dato7 = mensaje2.data[7];
30.
31.
32.                   //almacenamos la cadena de 32 bits
   correspondiente a los pids disponibles
33.
   bin32 = transformarcadena(dato3) + transformarcadena(dato4) + trans
   formarcadena(dato5) + transformarcadena(dato6);
34.              }
35.
36.          }
37.
38.      }
```

Figura 40. Función DescubrirPIDS()

Siguiendo una estructura similar a la introducida en el apartado 3.2.1 "Punto de partida y verificación del bus de comunicaciones", se genera en este caso la estructura *"mensaje"*, en la cual se establece el modo de operación 1 y se configura para realizar la petición al primer elemento del *array "pidsToRequest[]"*, el cual almacena las direcciones de los PID disponibles del vehículo. Este *array* contiene como primera posición la dirección 0x00, que devolverá los primeros 32 *bits* de disponibilidad.

Se ha modificado la temporalización de las peticiones, es decir, el *firmware* analizado en la sección 3.2.1 "Punto de partida y verificación del bus de comunicaciones" basaba la temporalización en una función automática que alternaba entre las funciones de petición y de recepción. Sin embargo, en este caso se establece que la propia función *"DescubrirPIDS()"* realice la petición y, tras un *delay* de 600 milisegundos, comience a filtrar los mensajes del bus buscando una trama de respuesta del vehículo, a la petición realizada.

En caso de que coincida la ID de respuesta esperada y el campo PID que corresponde al segundo *byte* recibido, el sistema se ocupa de almacenar los 8 *bytes* de información recibidos. Una vez recibida la información, se ejecuta la función *"transformarcadena()"* para elaborar una cadena binaria de 32 *bits* en forma de *String*, y refrescar la variable *"bin32"* que la almacena.

PeticionPID() - Petición de información

En caso de que el usuario haya seleccionado la opción "2", la función *"PeticionPID()"*, cuyo código se muestra en la Figura 41, se ocupa de gestionar la cadena que corresponde con la petición de información.

```
1. void PeticionPID(int PID) {
2.     bin32 ="";
3.     CANMessage mensaje;
4.     mensaje.id = ID_broadcast; //Id del tipo de mensaje
5.     mensaje.len = 8; //longitud del mensaje (caracteristica del
   objeto message)
6.     mensaje.data[0] = 0x02;
7.     mensaje.data[1] = OBD_PID_SERVICE; //modo de operacion
8.     mensaje.data[2] = PID; //pid objetivo
9.     carloop.can().transmit(mensaje);
10.          delay(400);
11.          bool respuesta = false;
12.
```

```
13.          CANMessage mensaje2;
14.          while(carloop.can().receive(mensaje2)) {
15.
     if (mensaje2.id >= ID_respuesta_minima && mensaje2.id <= ID_respu
   esta_maxima && mensaje2.data[2] == PID) {
16.
17.                   respuesta = true;
18.                   dato0 = mensaje2.data[0];
19.                   dato1 = mensaje2.data[1];
20.                   //PID code
21.                   dato2 = mensaje2.data[2];
22.                   //datos
23.                   dato3 = mensaje2.data[3];
24.                   dato4 = mensaje2.data[4];
25.                   dato5 = mensaje2.data[5];
26.                   dato6 = mensaje2.data[6];
27.                   //no tiene informacion de interes
28.                   dato7 = mensaje2.data[7];
29.
```

Figura 41. Función PeticionPID()

Con este propósito se ha creado la función *"PeticionPID()",* de manera similar a la introducida en el apartado 3.2.1 "Disponibilidad de PIDs y peticiones de información". Para realizar la consulta al PID seleccionado por el usuario, se ha generado la función con un argumento que sirva para introducir la dirección del PID objetivo de la petición.

Esta dirección de PID introducida se utiliza a modo de dirección de petición y como dirección de filtrado en la recepción de la respuesta. La temporalización sigue el esquema planteado en la función *"DescubrirPIDS()"* en el cual se utiliza un *delay* antes de filtrar las respuestas del bus.

Una vez se ha almacenado la respuesta proveniente del vehículo, se ha codificado una función *switch* que permita traducir la información que transmite el coche, según los parámetros de decodificación vistos en el apartado 2.3.4 "Decodificación de las tramas de respuesta".

```
1.   Switch( PID) {
2. case 12: {PeticionPID(12);
3.             Serial.println("Revoluciones del motor");
4.             float engine_rpm = ((256*dato3)+dato4)/4;
5.             Serial.println(engine_rpm);
6.       °     Serial.print("rpm");
7.   }
8.             break;
9. }
```

Figura 42. Switch del firmware

En el fragmento de código mostrado en la Figura 42 se muestra un ejemplo de dicha codificación. En caso de haber realizado la petición a la dirección correspondiente a las revoluciones del motor (dirección 0x0C), el sistema realiza la operación $\frac{dato4}{4} + (256 * dato3)$, que corresponde con la interpretación de los *bytes* recibidos, siendo respectivamente el *"dato3"* y el *"dato4"* los *bytes* 3 y 4 de la respuesta. Una vez realizada la interpretación, se muestra por pantalla vía serie el resultado de la operación.

3.5 Corrección y validación

3.5.1 Punto de partida y verificación

Con objetivo de comprobar el correcto funcionamiento del bus de comunicaciones y crear un punto de partida para la creación del *firmware* en el dispositivo *Photon*, se ha procedido a realizar la prueba con el código analizado en el capítulo 3.2.1 "Punto de partida y verificación del bus de comunicación"

En la Figura 43 se puede apreciar la interfaz de usuario de la aplicación utilizada en el ordenador portátil, correspondiente al software *PuTTY* [19]. Esta aplicación está configurada de forma que sea compatible con las dos interfaces serie que se utilizan en los dispositivos *Photon* y *RedBear Duo,* estableciendo una velocidad de transmisión de 9600 baudios y usando el puerto serie virtual COM4.

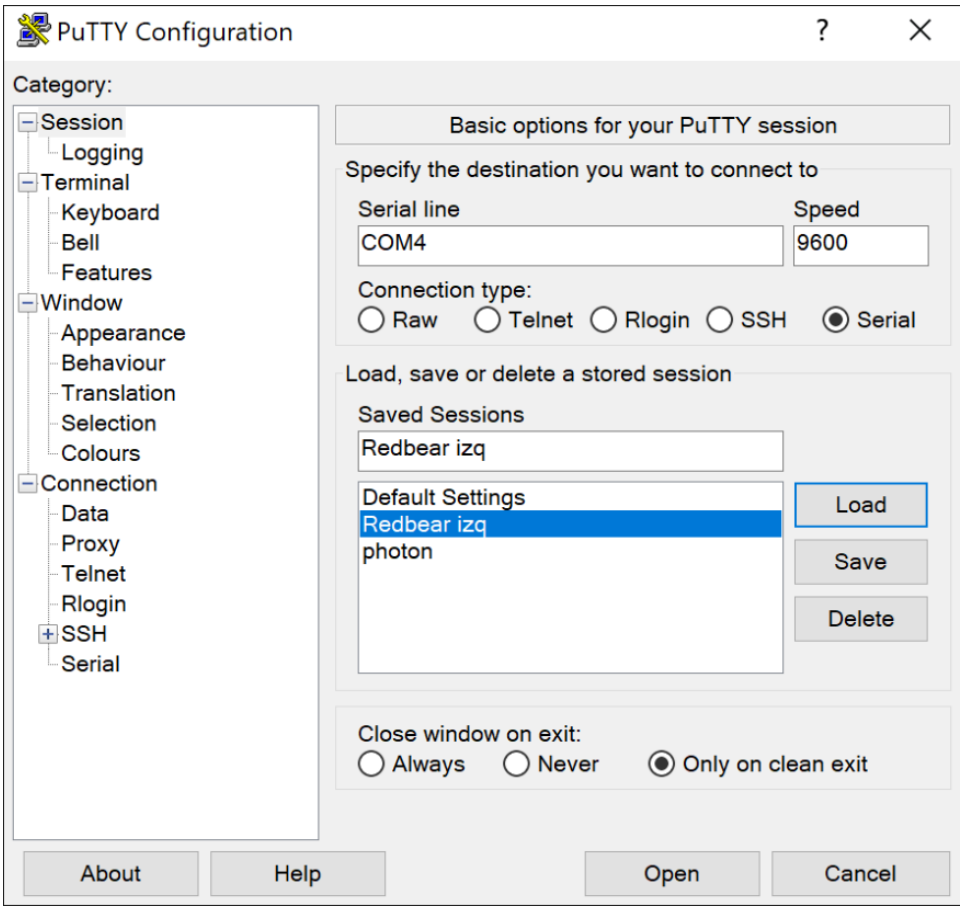

Figura 43. Interfaz PuTTY

Como se apreció en el análisis de los fragmentos del código del *firmware* del dispositivo *Photon,* éste se encargará de realizar peticiones con la dirección PID definida para determinar la velocidad del vehículo. Estas peticiones generarán una respuesta por parte del vehículo que el código debe ser capaz de filtrar.

Además de realizar la labor de filtrado de los paquetes CAN, se ha incluido dos representaciones vía puerto serie que se ocuparán de mostrar por pantalla todos los ID *(mensaje.id)* y la dirección PID (segundo *Byte* del vector datos) característicos de las todas las tramas CAN que envíe el vehículo.

Partiendo del objetivo de comprobar el funcionamiento del bus de comunicación, se han utilizado diferentes temporalizaciones en las funciones encargadas de enviar y recibir una trama CAN (“*EnviarPeticion()*” y “*esperarRespuesta()*”) y se ha utilizado el ID *broadcast* asociado a la transmisión de mensajes, mientras que se ha usado un único ID de respuesta para el filtrado. Con esto se espera realizar una doble verificación del

funcionamiento, ya que se demuestra, por un lado, la correcta temporalización, y por otro, una respuesta con una ID homogénea ante ID de petición genéricos.

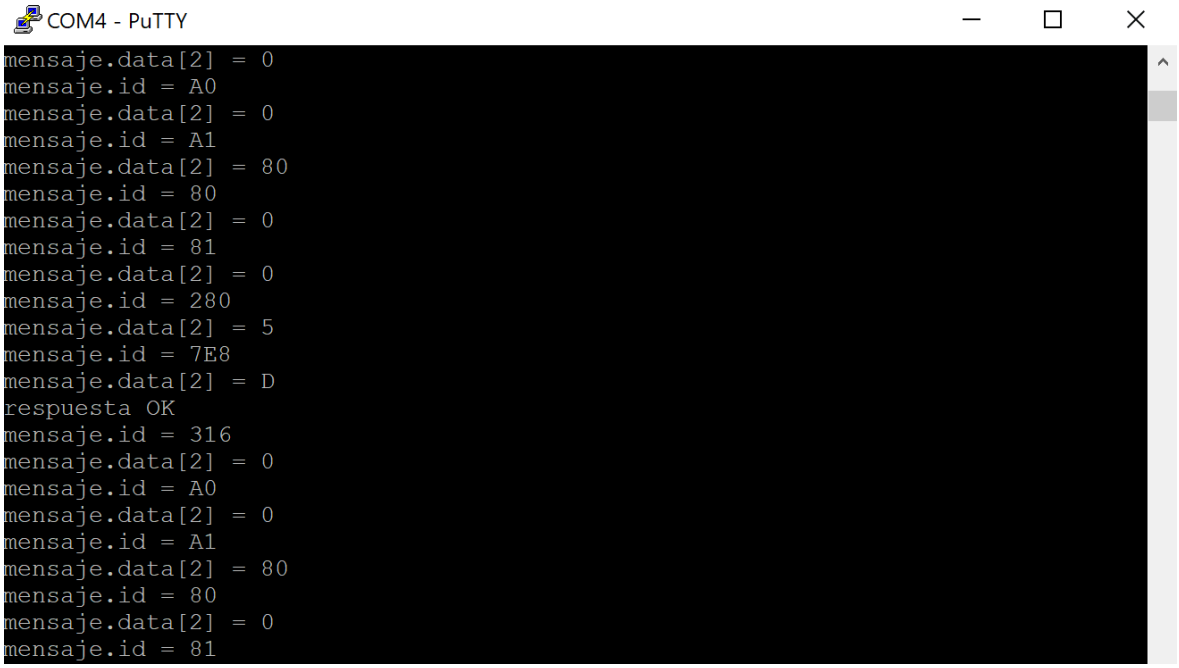

Figura 44. Pantalla PuTTY verificación del bus

Como se puede apreciar en la Figura 44, el dispositivo *Photon* muestra vía serie todos los ID y las direcciones PID de los paquetes que recibe vía bus CAN. Asimismo, se demuestra que ha realizado correctamente la labor de filtrado por ID de respuesta ya que ha entrado en el condicional que permite mostrar el mensaje de verificación *"Respuesta OK"*.

Aunque aún no se ha realizado ningún tipo de operación con la información, se puede afirmar que el bus de comunicaciones funciona correctamente y el código asociado a esta prueba servirá como marco para añadir las funciones posteriores.

3.5.2 Disponibilidad de PID y peticiones de información

Para esta prueba se ha utilizado el código analizado en la sección 3.2.2 "Disponibilidad de PIDs y peticiones de información".

En la Figura 43 se indica el menú contextual que muestra la interfaz serie cuando el dispositivo *Photon* comienza a funcionar. En este menú, el usuario será capaz de conocer qué direcciones PID están disponibles en el vehículo (opción 1), además de poder realizar una petición única insertando un número binario de 32 bits en el cual se refleje con un valor 1 lógico el PID del cual se desea conocer la información (opción 2).

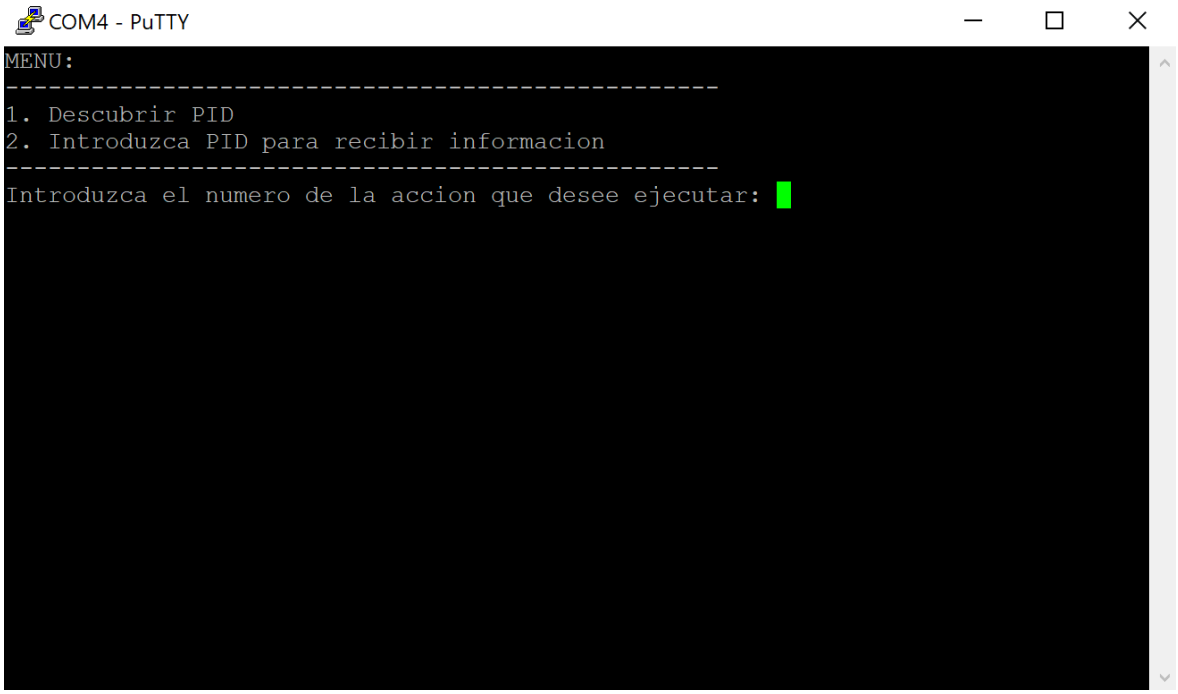

Figura 45. Menú inicial

En un primer caso práctico, el usuario pulsará la tecla 1 a fin de conocer qué PID tiene disponibles, como se muestra en la Figura 45.

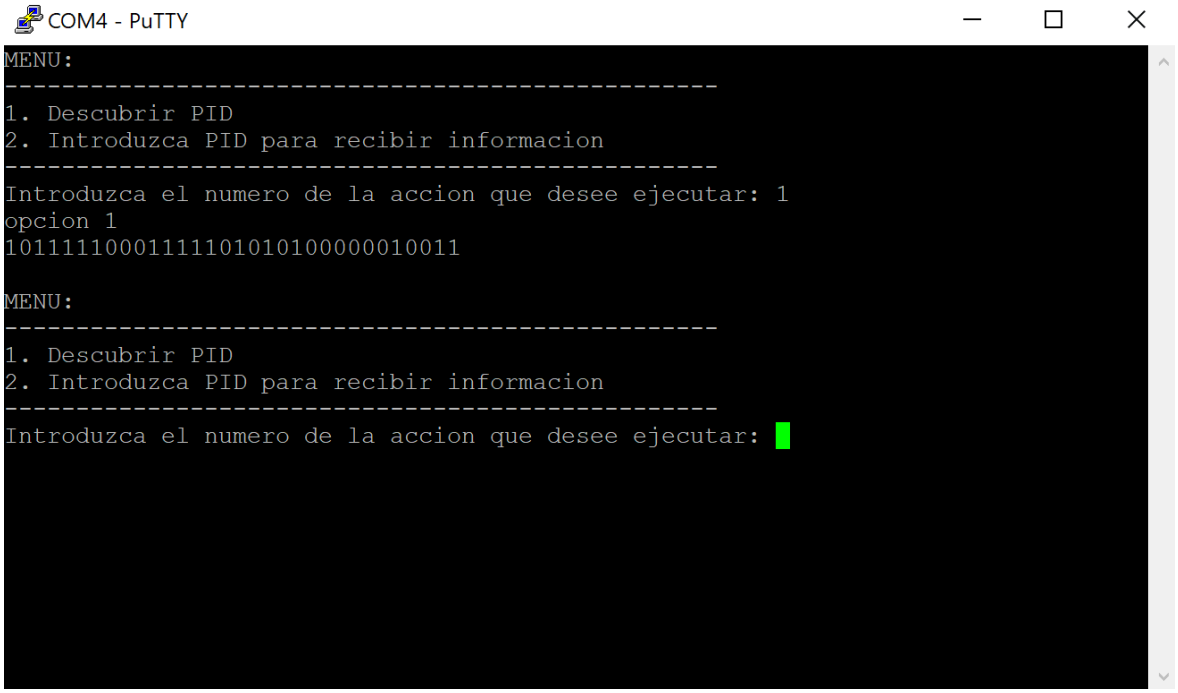

Figura 46. Opción 1 DescubrirPIDS()

Como se aprecia en la Figura 46, al seleccionar la opción 1 el sistema realiza la consulta de los PID disponibles dentro del primer *bitmap* del sistema. Esta función, que está diseñada para obtener únicamente el primer *bitmap* de disponibilidad, servirá de precursor para la función implementada en el sistema final, que si contempla todos los *bitmaps* disponibles del vehículo.

Una vez que el sistema interpreta la orden del sistema, realiza la consulta a la dirección 0x00. El código binario que aparece en la Figura 46 indica con un valor 1 lógico las direcciones PID disponibles. Del mismo modo, las direcciones no disponibles para realizar consultas están señaladas con un valor 0 lógico.

Después de obtener dicha cadena binaria, se procede a comprobar su validez realizando dos comprobaciones aleatorias de PID que devuelven una información que permita verificar, de manera práctica, el correcto funcionamiento de la función que implementa el descubrimiento de PID disponibles *(DescubrirPIDS())* y la encargada de realizar consultas a PID concretos *(PeticionPID()).*

La primera petición realizada ha sido con la dirección definida para determinar las revoluciones del motor, que corresponde con la dirección asociada a la posición del calor

“1” lógico en la cadena binaria introducida. Está petición conllevará una respuesta que será procesada por el código analizado en el capítulo 3.3.2 “Disponibilidad de PIDs y peticiones de información”.

Para que el usuario pueda llevar a cabo esta petición, el sistema requerirá que se introduzca un “1” lógico que, en este caso, corresponde con la decimosegunda posición del vector de 32 *bits* introducido. Este proceso se puede observar en la Figura 47.

Figura 47. Opción 2 petición revoluciones del vehículo

Una vez que el sistema detecta una trama correcta, comienza a realizar las peticiones al PID requerido. El sistema filtrará todos los mensajes dentro del bus de comunicación en busca de aquel que dispone de la ID de respuesta requerida, y la dirección PID objetivo de la petición (0x0C en este caso). Si existen tramas que puedan superar la condición de filtrado, el *firmware* está preparado para interpretar la información que recibe en forma de 4 *bytes*, procesándolos como se indicó en el apartado 2.3.4 “Decodificación de las tramas de respuesta”.

Este parámetro ha sido intencionadamente elegido, ya que se puede comprobar y verificar la validez simplemente observando el propio cuadro de mandos del vehículo, como se muestra en la Figura 48.

Figura 48. Cuadro de mandos del vehículo de pruebas

El vehículo de pruebas, en ralentí, indicaba 1000 revoluciones por minuto en el momento de realización de la prueba del *firmware* del dispositivo *Photon.* Por tanto, se puede afirmar que el código ha conseguido realizar la petición de información a la dirección PID correspondiente a las revoluciones del motor, e interpretar correctamente la respuesta.

La otra petición se ha realizado hacia la dirección PID correspondiente a la temperatura del líquido refrigerante del motor. La respuesta del sistema vía serie se indica en la Figura 49.

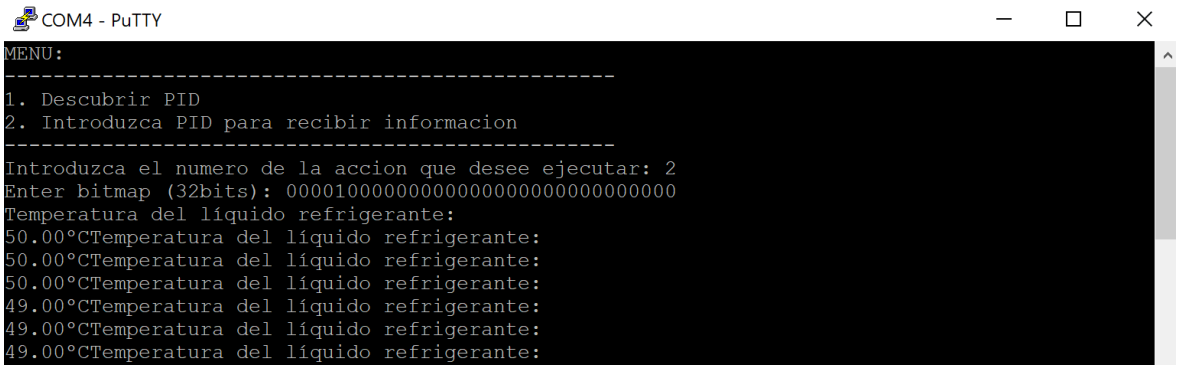

Figura 49. Opción 2 petición temperatura del refrigerante del vehículo

De manera análoga a la petición correspondiente a las revoluciones del motor, la información del refrigerante se puede observar en el área del cuadro que aparece en la Figura 50.

Figura 50. Temperatura exterior y del líquido refrigerante

En dicha área aparece la temperatura exterior (valor de 26ºC) y la temperatura del líquido refrigerante (media circunferencia con las numeraciones 50ºC y 90ªC). Se puede intuir que la temperatura base del líquido refrigerante de este vehículo durante su funcionamiento parte de los 50ºC, y llega hasta unos 90ºC de valor estándar de funcionamiento.

En la Figura 49 se aprecia que el valor de la temperatura es de unos 50ºC, lo cual se debe a que el motor se puso en marcha para la realización de la prueba, y no había tenido tiempo de aumentar la temperatura del circuito de refrigeración.

Igualmente, en la Figura 51 se indican los resultados obtenidos pasados unos minutos desde su puesta en marcha, con el fin de comprobar la homogeneidad de los resultados obtenidos.

```
56.00ºCTemperatura del líquido refrigerante:
56.00ºCTemperatura del líquido refrigerante:
57.00ºCTemperatura del líquido refrigerante:
57.00ºCTemperatura del líquido refrigerante:
```

Figura 51. Aumento de la temperatura con el paso del tiempo

Así, se pudo comprobar que la temperatura del líquido refrigerante había subido 7ºC en el transcurso del tiempo, por lo que se concluye que se está recibiendo la información correcta en tiempo real.

Después de haber comprobado el funcionamiento del *firmware* del dispositivo *Photon* realizando las pruebas relacionadas con las dos funciones elementales implementadas, se ha creado un marco de funcionamiento que se seguirá para la realización de la plataforma final *HW/SW* gracias a la elevada compatibilidad que existe entre los dispositivos IoT *Photon* y *RedBear Duo.*

4 Implementación de la interrogación vía BLE

4.1 Tecnología BLE

4.1.1 Introducción

la plataforma HW/SW a desarrollar en este Trabajo Fin de Grado se centra en el uso de la tecnología de comunicación *Bluetooth Low Energy* (BLE), basada en la especificación *Bluetooth* 4.0. Esta tecnología se conforma como una alternativa reducida y optimizada al protocolo estándar *Bluetooth*. Esta tecnología está orientada a crear una transmisión inalámbrica con el mínimo consumo de potencia posible, para la implementación de soluciones de bajo coste en sistemas limitados en ancho de banda y complejidad [22].

La especificación *Bluetooth 4.0* sigue una filosofía orientada a la transmisión de pequeñas cantidades de datos a corta distancia. No está ideado para mantener una conexión entre dispositivos por un largo periodo de tiempo transmitiendo grandes cantidades de datos a alta velocidad, como si lo está el estándar *Bluetooth* clásico. En consecuencia, los dispositivos BLE solo estarán activos cuando se les pide la transmisión de datos con el incremento de vida útil de la batería del dispositivo que esto supone.

En la Tabla 13 se indica la evolución de las tasas de transferencia en las diferentes especificaciones del estándar *Bluetooth*.

Tabla 13. Evolución tasas de transferencia Bluetooth

Versión	Tasa de transferencia	Nombre característico
1.2	721 kb/s	-
2.0 + EDR	3 Mb/s	Enhanced Data Rate (EDR)
3.0 + HS	24 Mb/s	High-Speed
4.0	1 Mbit/s – 2 Mbit/s	Bluetooth Low Energy (BLE)

Con el objetivo de reducir los costes de implementación de la tecnología BLE en dispositivos que posiblemente no dispongan de ninguna tecnología inalámbrica, se han tomado diferentes decisiones que permiten minimizar los costes de dicha implementación.

Siguiendo la filosofía de la tecnología BLE se ha apostado por el uso de baterías o pilas tipo botón, que por su tamaño, precio y disponibilidad se perfilan como la opción ideal para cumplir los requisitos de los sistemas basados en BLE (bajo coste, baja tasa de datos y bajo consumo) [23].

Esta tecnología, inicialmente apodada *wibree*, fue estandarizada por *Bluetooth Special Interest Grupo* (SIG), lo que ha permitido que el coste de licencia de los dispositivos *Bluetooth* se reduzca considerablemente, hecho que potencia su expansión.

Existen rangos de frecuencia que no resultan idóneos para la implementación de esta tecnología, como la banda 60 GHz, que no resulta atractiva debido al coste económico de su utilización o por problemas de estandarización que sufren, por ejemplo, las bandas de 800/900 MHz (diferentes frecuencias y normativas dependiendo de la localización). La utilización de la banda libre ISM (*Industrial, Scientific and Medical)* de 2.4 GHz ofrece una clara ventaja en costes, ya que no es necesario pagar una licencia para su utilización. Trabajar en la banda de 2.4 GHz ofrece también la capacidad de operar a nivel global con un impacto mínimo en costes y facilidad para manejar grandes volúmenes de facturación.

La banda de 2.4 GHz, al ser una banda libre, se ve afectada por un alto tráfico que puede conducir a la congestión de la banda. Debido a esto, fue necesario implementar un sistema capaz de trabajar con garantías en entornos con interferencias. Para tal fin se implementó la técnica *Adaptative Frequency Hopping*. Esta técnica permite no solo detectar fuentes de interferencias, sino evitarlas de forma adaptativa en el futuro que, junto con la recuperación efectiva de los paquetes perdidos durante la transmisión, hacen de esta tecnología *Bluetooth* un sistema de transmisión inalámbrica robusto.

4.1.2 Dispositivos en la red

Dentro de la especificación de *Bluetooth* se contemplan fundamentalmente las tecnologías *Bluetooth* clásica y *Bluetooth Low Energy* [25]. Sin embargo, solo los dispositivos que implementen la versión 4.0, o superior, serán compatibles con los que implementen *Bluetooth Low Energy*.

Atendiendo a esta diferenciación, existen dos tipos de implementaciones de la tecnología BLE en los dispositivos:

- ***Dual-mode:*** Dispositivos que soportan BLE y *Bluetooth Classic*.
- ***Single-mode:*** Dispositivos que solo soportan BLE.

En la Tabla 14 se indica una relación entre los modos de operación y la posible compatibilidad entre ellos.

Tabla 14. Comunicación entre modos de operación Bluetooth

Modo	Single-Mode	Dual-Mode	Clásico
Single-Mode	LE	LE	No disponible
Dual-Mode	LE	Clásico	Clásico
Clásico	No disponible	Clásico	Clásico

4.1.3 Características básicas del protocolo

Velocidad de transmisión

La velocidad de transmisión del dispositivo en muchos casos responde a términos ambientales, lo cual determina el límite de velocidad a la que un dispositivo BLE puede transmitir. Normalmente esta velocidad se ve reducida debido a los factores tales como tráfico bidireccional, sobrecarga del protocolo, limitaciones *hardware* del sistema, etc. Esta velocidad, como se analizaba en la Tabla 13, típicamente podrá oscilar entre 1 y 2 Mbps.

Longitud de trama de datos

En lo referente a los paquetes de datos intercambiados bidireccionalmente entre un dispositivo *Master* y un dispositivo *Slave* compatibles con la tecnología BLE 4.0 durante los eventos de conexión, estos tienen una carga útil de 27 *bytes*, pero los protocolos superiores de la pila de protocolos de BLE limita la cantidad real de datos de usuario a 20 *bytes.*

Rango de operación

El rango de disponibilidad de cualquier sistema inalámbrico se ve afectado por multitud de factores, desde la distribución de los objetos del entorno, hasta la orientación

del dispositivo. Este protocolo, en cambio, ha sido desarrollado para ser utilizado en distancias muy reducidas.

La potencia de transmisión se establece en un rango comprendido entre -30 dBm y 0 dBm, buscando un valor de compromiso que, aportando un rango práctico de uso, no represente un impacto significativo en el consumo de batería del dispositivo.

Con el objetivo de incrementar la vida útil de la batería, típicamente se establece un rango de operación comprendido entre 2 y 5 metros de distancia.

Mecanismo de descubrimiento entre los dispositivos Peripheral y Central

En este Trabajo Fin de Grado se dispone de dos dispositivos que intervienen en el intercambio de información con el vehículo, los cuales son el dispositivo IoT *RedBear Duo,* que actuará con el rol de dispositivo *Peripheral* para conectarse con el terminal móvil, que actuará con el rol de dispositivo *Central.*

En la especificación de BLE, la capa de enlace (*LL, Link Layer)* es la parte responsable de controlar, negociar y establecer los enlaces [23][25], seleccionar las frecuencias para la transmisión de datos, admitir diferentes topologías y dar soporte a diversas formas de intercambio de datos. En consecuencia, la capa de enlace es la responsable de los procesos de *Advertising y Scanning*, así como de la creación y mantenimiento de las conexiones y de la estructura de los paquetes de datos transmitidos.

El funcionamiento de la capa de enlace puede asemejarse al de una máquina de estados en la que figuran los siguientes estados:

- *Standby:* Este es el estado predeterminado, no se reciben ni se transmiten paquetes. Un dispositivo puede posicionarse en este estado desde cualquier otro estado posible.
- *Advertising:* Los dispositivos *Peripheral* transmiten paquetes de *Advertising* en los canales de *Advertising.* En este estado también se estará a la escucha de los posibles dispositivos *Central* que responden a los paquetes de *Advertising,* y se les responderá en consecuencia. Para acceder a este estado, el dispositivo deberá

encontrarse primero en el estado *Standby*, y cambiará de estado cuando la capa de enlace inicie el proceso de *Advertising*.

- *Initiating:* La capa de enlace recibe los paquetes desde el dispositivo *Advertiser* y responde iniciando una conexión. Un dispositivo *Central* entrará en este estado antes de pasar al estado correspondiente a la conexión. A este estado se puede llegar desde el estado *Standby* cuando el dispositivo *Scanner* decide iniciar una conexión con el dispositivo *Advertiser,* actuando como dispositivo *Initiatior.*
- *Scanning:* El dispositivo que se encuentre en este estado escucha los paquetes de *Advertising* enviados por el dispositivo que se encuentre en el estado *Advertiser* a través de los canales asignados, siendo posible que se solicite información adicional para explorar los dispositivos BLE existentes. Un dispositivo puede llegar a este estado desde el estado *Standby*, cuando la capa de enlace comienza el proceso de *Scanning*.
- *Connection:* Este estado se alcanza cuando el dispositivo *Central* está conectado a otro dispositivo *Peripheral,* definiéndose los roles de *Master y Slave*. A este estado se puede llegar a través del estado *Initiating* o desde el estado *Advertising.* Si el dispositivo llega al estado *Connection* desde el estado *Initiating*, el dispositivo actúa como *Master*, mientras que cuando se llega desde el estado *Advertising*, el dispositivo se comporta como *Slave*.

En la Figura 52 se indica un modelo de máquina de estados que refleja la relación entre los diferentes estados de la capa de enlace del dispositivo que implementa BLE

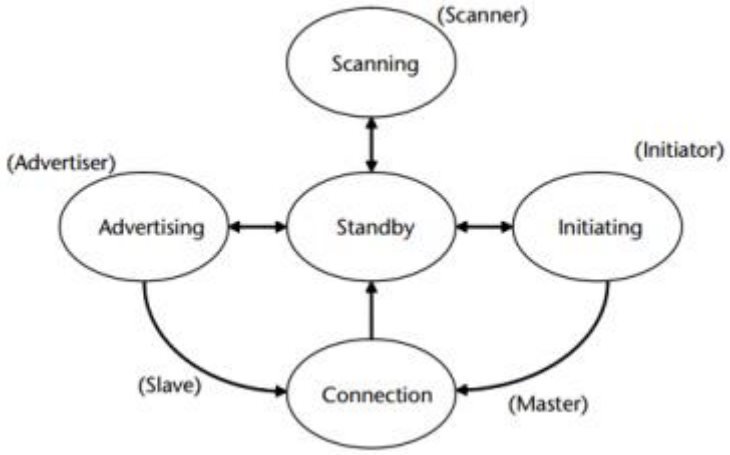

Figura 52.Modelo máquina de estados capa de enlace

4.1.4 Topología de la red

Los dispositivos basados en la tecnología BLE disponen de dos formas principales de comunicación con el medio [25], a través de *Broadcasting*, o mediante conexiones punto a punto. Cada mecanismo conlleva una serie de ventajas y limitaciones que los hacen idóneos para diferentes aplicaciones. Ambos mecanismos están sujetos a las pautas establecidas por el perfil *Generic Access Profile* (GAP), analizado posteriormente.

Broadcast

El mecanismo *Broadcast* se basa en la filosofía de enviar la información o tramas a todos los integrantes de la red que se encuentren en el rango de operación. En la Figura 53 se indica un ejemplo de la topología *broadcast* empleando como medio transmisor un dispositivo con la tecnología BLE.

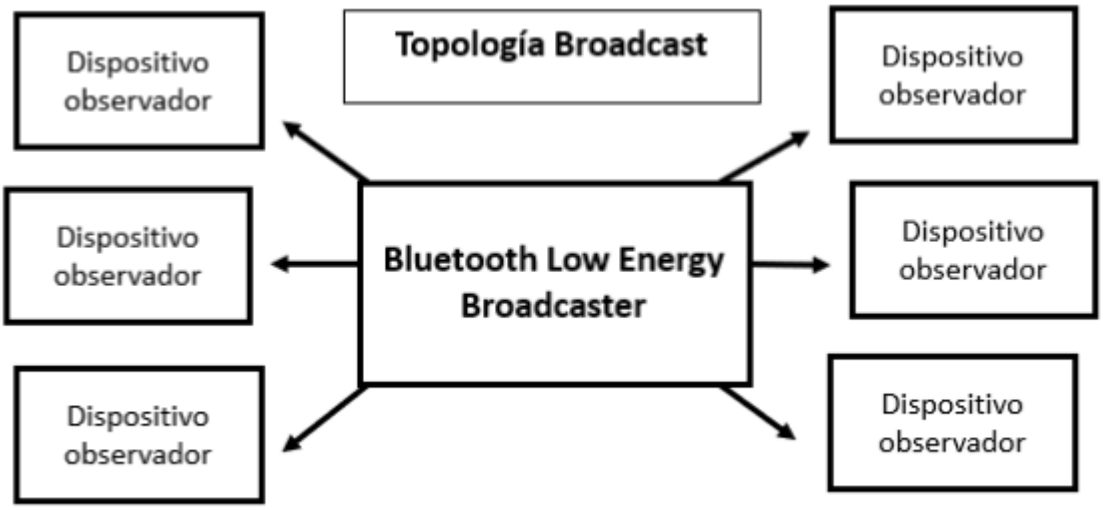

Figura 53. Topología broadcast

El dispositivo *Broadcaster* enviará paquetes de *Advertising*, no conectables y de forma periódica, a cualquier dispositivo en escucha. Por otro lado, el dispositivo que actúe como observador realizará escaneos periódicos de las frecuencias preestalecidas para recibir y procesar cualquier paquete de *Advertising* que se esté transmitiendo en ese momento. Estos paquetes de *Advertising* transportan un estándar *de 31 bytes* (*Payload)*. En caso de no ser suficiente, BLE dispone de un paquete de *Advertising* secundario y opcional que añade otros 31 *bytes.*

Este mecanismo es de especial utilidad si se desea transmitir datos a más de un dispositivo a la vez, en BLE. Es un método rápido, siendo adecuado cuando se desea trasmitir una pequeña cantidad de datos en un tramo temporal determinado, o para múltiples dispositivos. Lamentablemente, no ofrece garantías en términos de seguridad o privacidad por lo que hay que limitar la información sensible que se envía.

Conexiones punto a punto

Bajo la situación de crear una comunicación bidireccional, o la necesidad de enviar más datos de los que pueden soportar los dos *Advertising Payload*, se crea una conexión directa entre dos dispositivos BLE. De esta manera, se crea una vía privada de comunicación entre dos dispositivos, que aporta mayor grado de seguridad y privacidad.

Para comenzar la conexión, el dispositivo *Central* escucha y recoge los paquetes de *Advertising* de los posibles dispositivos *Peripheral* a su alcance. Una vez detectado el dispositivo *Peripheral* emisor de los paquetes de *Advertising*, el dispositivo *Central* enviará una solicitud para establecer una conexión privada entre ambos y comenzará el intercambio de datos.

En la Figura 54 se representa un ejemplo de la topología punto a punto BLE.

Figura 54. Topología punto a punto

Desde la versión 4.1 de la especificación de *Bluetooth Low Energy*, se eliminan las restricciones en cuanto a las combinaciones de roles: Cualquier dispositivo puede actuar, tanto como *Central*, como *Peripheral*, además de que un dispositivo *Central* puede disponer de más de una conexión activa.

Una ventaja de las conexiones directas es la capacidad que permiten de organizar los datos con un control mucho más preciso de cada campo o propiedad, gracias al uso de capas de protocolo adicionales y, más concretamente, al *Generic Attribute Profile* (GATT), en el que los datos se organizan en unidades denominadas servicios y características.

Las conexiones directas presentan una buena predisposición para utilizar técnicas de ahorro de energía, ya que permite extender el retraso entre eventos de la conexión, o enviar datos solo cuando hay nuevos valores disponibles, en lugar de emitir datos continuamente sin conocer que dispositivos están a la escucha y con qué frecuencia muestrean la búsqueda. Conocer de antemano cuándo se producirán los eventos de

conexión también permite desactivar la búsqueda durante un mayor tiempo lo que repercutirá en un mayor ahorro de energía.

Por último, destacar que las topologías basadas en *Broadcasting* y conexiones punto-a-punto son combinables, como se aprecia en la Figura 55.

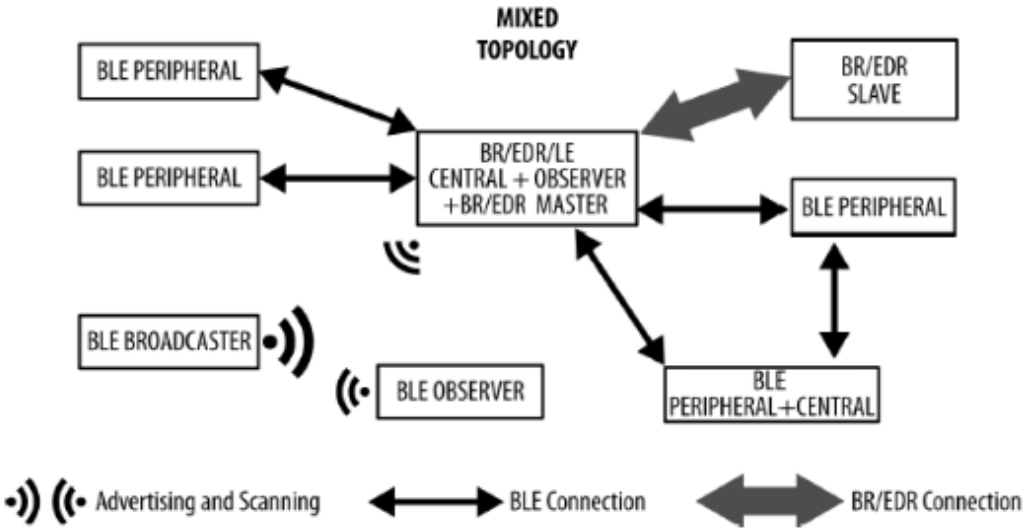

Figura 55. Topología mixta

4.1.5 Protocolos y perfiles

En este apartado se procederá a analizar las diferencias, dentro de la especificación *Bluetooth*, entre los conceptos protocolo y perfil entre los que, desde los inicios de la especificación, existe una separación bien definida.

El concepto protocolo responde a los bloques utilizados por todos los dispositivos que entran en la especificación *Bluetooth.* Se compone de las capas que implementan los diferentes formatos de paquete, encaminamiento, multiplexación, codificación y decodificación, que permiten que los datos se envíen de manera efectiva entre dispositivos.

Los perfiles, por otro lado, se corresponden con las divisiones verticales de funcionalidad que cubren los modos básicos de operación requeridos por todos los dispositivos (GAP, GATT) o usos en casos específicos. Cabe destacar que, dentro de un dispositivo que utilice la tecnología BLE, no existe un perfil propiamente dicho, sino que un perfil se compone de

una colección de servicios predefinidos por parte de la *Bluetooth* SIG, o por el propio fabricante del dispositivo.

En los próximos apartados se analizarán los perfiles genéricos GAP y GATT que servirán para configurar la funcionalidad de los dispositivos IoT.

4.1.6 Generic Access Profile (GAP)

Generic Access Profile es la capa de control de mayor nivel de abstracción de la tecnología BLE [25]. Este perfil debe estar presente en todos los dispositivos que implementen BLE, ya que permite que interactúen entre ellos.

El perfil GAP establece una serie de normas que cualquier implementación BLE debe seguir para permitir que los diferentes dispositivos sean capaces de comunicarse y realizar otras operaciones dentro de un estándar.

Para analizar de manera más clara este perfil, es necesario definir ciertos aspectos sobre la interacción de los dispositivos dentro de una red basada en BLE.

- **Modo:** Corresponde al estado al que el dispositivo puede cambiar durante un periodo de tiempo para permitir utilizar las funciones asociadas a dicho estado.
- **Rol:** Un dispositivo que utiliza la tecnología BLE es capaz de operar en uno o más roles al mismo tiempo. Un rol define las restricciones y pautas de actuación establecidas. El perfil GAP es el encargado de establecer las posibles interacciones de estos roles.
- **Procedimiento:** Conforma la secuencia de acciones que permiten a un dispositivo realizar acciones concretas. Estos procedimientos típicamente están asociados a modos de operación concreto.

Roles definidos por el perfil GAP

El perfil GAP define cuatro roles posibles para los dispositivos que implementen la tecnología BLE:

- ***Broadcaster:*** Ideado para aplicaciones en las que se realicen transmisiones de datos de forma periódica. Este rol se basa en el envío por parte del dispositivo de la información incluida dentro de los paquetes de *Advertising.*
- ***Observer:*** Pensado para las aplicaciones que solo requieran la recepción de datos. El dispositivo que actúe con el rol de *Observer* será el encargado de escuchar en busca de datos en los paquetes de *Advertising* enviados por los dispositivos *Broadcaster.*
- ***Central:*** Este rol corresponde a la capa de enlace del dispositivo *Master*, el cual debe ser un dispositivo con la capacidad, en términos de potencia, de iniciar y establecer múltiples conexiones. El dispositivo *Central* será el encargado de buscar los paquetes de *Advertising* de otros dispositivos con el objetivo de iniciar la conexión con un dispositivo en la red.
- ***Peripheral:*** El rol *Peripheral* pertenece a la capa de enlace del dispositivo *Slave* de la red. El dispositivo que tenga este rol será el encargado de enviar paquetes de *Advertising* a la espera de que los dispositivos de tipo *Central* los encuentren, lo que permite que el dispositivo *Peripheral* realice una conexión directa con el dispositivo *Central*.

4.1.7 Generic Attribute Profile (GATT)

Generic Attribute Profile (GATT) es el perfil que define cómo se intercambian todos los datos de perfil y de usuario a través de una conexión BLE. A diferencia del perfil GAP, que especifica las interacciones a bajo nivel, el perfil GATT abarca los procedimientos y formatos de transferencia de datos.

Este perfil proporciona una referencia para todos los perfiles basados en GATT, que cubre usos concretos y permite la interoperabilidad entre dispositivos de diferentes proveedores. Por lo tanto, todos los perfiles BLE estándar se basan en GATT y deben seguir las directrices definidas para funcionar correctamente. GATT se perfila así, como una parte clave en la especificación BLE, ya que cada elemento de datos significativos para las aplicaciones y usuarios deben formatearse, empaquetarse y enviarse según sus directrices.

El perfil GATT utiliza el *Attribute Protocol* (ATT), junto con su protocolo de transporte, para intercambiar datos entre dispositivos. Estos dispositivos están organizados jerárquicamente en secciones llamadas servicios, que agrupan piezas de datos de usuario llamadas características.

Attribute Protocol (ATT)

En BLE cada dispositivo se puede comportar como cliente, servidor o ambos a la vez, sin dependencia de si es *Master* o *Slave*. El protocolo también se caracteriza por una estricta secuenciación, si una solicitud aún está pendiente, no se pueden enviar más solicitudes hasta que la respuesta se reciba y se procese.

Cada servidor contiene datos organizados en forma de atributos, a cada uno de los cuales se le asigna un identificador de atributo de 16 *bits*, un identificador único universal (UUID), un conjunto de permisos y un valor. El UUID especifica el tipo y la naturaleza de los datos contenidos en el valor.

Cuando un cliente intente leer o escribir valores de atributos desde o hacia un servidor, emite una solicitud de lectura o escritura al servidor con el identificador. El servidor devolverá el valor del atributo o una señal de recepción. Si en cambio se va a realizar una operación de escritura, se espera que el cliente proporcione datos que sea consistentes con el tipo de atributo.

Roles en el perfil GATT

El perfil GATT, al igual que el perfil GAP, dispone de una serie de roles definidos que los dispositivos pueden utilizar, estos son cliente y servidor:

- **Servidor:** El servidor GATT corresponde al servidor del protocolo ATT. Es el encargado de recibir las solicitudes de un cliente y devolver las respuestas adecuadas, además de almacenar y poner a disposición del cliente los datos del usuario organizados en atributos. Cada dispositivo BLE debe incluir, al menos, un servidor GATT básico que pueda responder a las solicitudes de los clientes.
- **Cliente:** El cliente GATT corresponde al cliente del protocolo ATT. Es el encargado de enviar solicitudes a un servidor y recibir las respuestas de éste. El primer paso que seguirá el cliente será el descubrimiento de la presencia y la naturaleza de los

atributos presentes en el servidor. Una vez realizado este descubrimiento el cliente es capaz de leer y escribir atributos que se encuentren en el servidor.

Cabe destacar que la utilización de los roles dentro del perfil GATT es independiente del rol o roles GAP utilizados. Esto proporciona, tanto a un dispositivo con el rol *Central* GAP, como a un dispositivo con el rol *Peripheral* GAP, la posibilidad de actuar como un cliente o servidor GATT.

Universally Unique Identifier (UUID)

El UUID es identificador compuesto por un número de 128 bits (16 *Bytes*) único a nivel global. Estos identificadores se utilizan en muchos protocolos y aplicaciones que no están relacionadas con la tecnología *Bluetooth*. Las reglas de formato, uso y generación están definidas en ISO / IEC 9834-8: 2005.

El SIG proporciona identificadores únicos (UUID) para todos los tipos, servicios y perfiles que define y especifica. En caso de que una aplicación necesite su propio identificador debido a que necesite implementar un uso no definido por el estándar o porque los identificadores ya definidos no cumplen los requisitos exigidos, se pueden generar estos identificadores en la página de ITU.

Atributos

Los atributos conforman la entidad de datos más pequeña definida por GATT. Se componen de fragmentos direccionables de información que pueden contener datos de usuario o metadatos. GATT puede operar únicamente con atributos, por lo que para que los clientes y servidores interactúen la información debe estar estructurada. Los atributos siempre se ubican en el servidor, mientras que el cliente es el que se ocupa de acceder a ellos y modificarlos. Dado que la especificación *Bluetooth* define los atributos conceptualmente, no hay obligación de utilizar un formato o mecanismo de almacenamiento interno concreto. Cada atributo contiene información sobre sí mismo y sobre los datos actuales de la forma que se describe a continuación:

- **Identificador:** Corresponde al identificador único de 16 *bits* para cada atributo en un servidor GATT. Este identificador no variará entre transacciones o a través de las conexiones, lo que permite el direccionamiento de estos. El valor 0x0000 refleja un identificador no válido, y un servidor GATT dispone de hasta 0xFFFE (65535) identificadores. El cliente es el encargado de obtener los identificadores de los atributos que le interesan por medio de la función de descubrimiento.
- **Tipo:** Hace referencia al tipo del atributo, y se compone de un UUID de 16, 32 o 128 *bits* que define el tipo de datos presentes en el valor del atributo.
- **Permisos:** Los permisos son metadatos que definen qué operaciones de ATT se pueden ejecutar en cada atributo particular y con qué requisitos de seguridad específicos. Estos permisos de denominan permisos de acceso, los cuales determinan si el cliente puede leer o escribir (o ambos) un valor de atributo. Cada atributo puede tener uno de los siguientes permisos de acceso:
    - ***None*:** El atributo no puede ser leído ni escrito por un dispositivo cliente.
    - ***Readable*:** El atributo puede ser leído por un dispositivo cliente.
    - ***Writable:*** El atributo puede ser escrito por un dispositivo cliente.
    - ***Readable and writable:*** El atributo puede ser leído y escrito por el dispositivo cliente.

  Además de los permisos de acceso, GATT define si se requiere algún tipo de cifrado para que el cliente pueda acceder al atributo, así como la posible necesidad de obtener el permiso del usuario para acceder a un atributo.

- **Valor:** El valor contiene los datos actuales asociados al atributo. No existen limitaciones sobre el tipo de datos que puede contener, aunque su longitud máxima está limitada a 512 *bytes*, de acuerdo con la especificación *Bluetooth.* Esta es la parte del atributo que un cliente puede, con los permisos adecuados, acceder para leer o modificar.

Jerarquía de datos y atributos

ATT funciona en términos de atributos y se basa en todos los conceptos analizados anteriormente para proporcionar una serie de unidades de datos de protocolo (PDU, *Protocol Data Unit)* comúnmente conocidas como paquetes, que permiten a un cliente acceder a los atributos de un servidor. GATT establece una jerarquía estricta para organizar los atributos de forma reutilizable y práctica. Esto permite el acceso y la recuperación de información entre el cliente y el servidor siguiendo un conjunto de reglas que conforman un estándar utilizado por todos los perfiles basados en GATT.

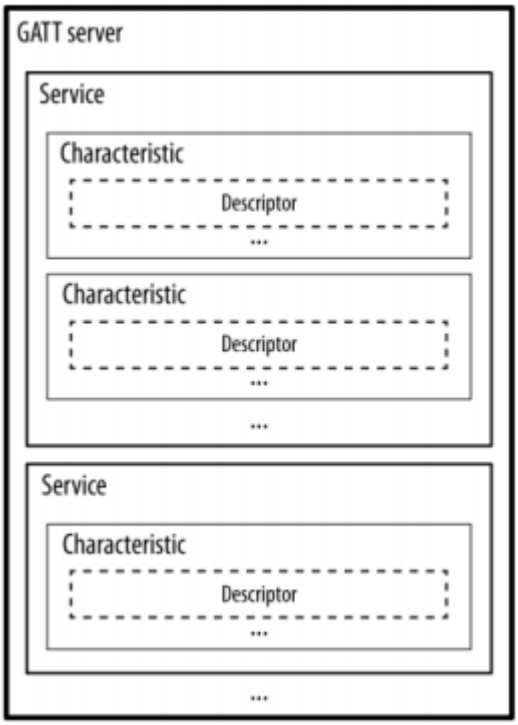

Figura 56. Jerarquía de datos y atributos de GATT

Como se aprecia en la Figura 56, Un servidor GATT agrupa los atributos en servicios, cada uno de los cuales pueden contener características. Estas características pueden incluir, a su vez, descriptores. Cualquier dispositivo BLE responde a esta jerarquía, lo que supone que todos los atributos en un servidor GATT están incluidos en una de estas tres categorías.

Servicios

Todos los atributos dentro de un solo servicio se conocen como la "Definición de servicio". En consecuencia, los atributos de un servidor GATT se componen con una sucesión de definiciones de servicio, cada una con un único atributo que marca el

comienzo de un servicio, llamado “Declaración de servicio”. En la Figura 57 se muestra un ejemplo de Declaración de servicio.

Handle	Type	Permissions	Value	Value length
0xNNNN	$UUID_{primary\ service}$ or $UUID_{secondary\ service}$	Read Only	Service UUID	2, 4, or 16 bytes

Figura 57. Declaración y valor de una característica

Tanto el UUID del servicio primario como el del servicio secundario se corresponden con estándares asignados por el SIG que se usan como un tipo exclusivo para introducir un servicio. Un servicio primario es el tipo estándar de servicio de GATT, mientras que el servicio secundario solo tiene sentido como modificador de servicios primarios y su utilización es escasa.

Características

En relación con la tecnología BLE, se puede definir una característica como un contenedor para los datos de usuario. Debe incluir al menos dos atributos: La Declaración de característica (compuesta por un conjunto de metadatos sobre los datos de usuario actuales) y el Valor de característica (compuesta por un atributo completo que contiene los datos de usuario en su campo de valor).

El Valor de característica, a su vez, puede ir seguido de descriptores que amplían aún más los metadatos contenidos en la Declaración de característica. La declaración, el valor, y los descriptores, forman la Definición de característica, que es el conjunto de atributos que conforman una característica específica.

Handle	Type	Permissions	Value	Value length
0xNNNN	$UUID_{characteristic}$	Read only	Properties, value handle (0xMMMM), characteristic UUID	5, 7, or 19 bytes
0xMMMM	Characteristic UUID	Any	Actual value	Variable

Figura 58. Declaración y valor de una característica

El tipo del atributo de declaración de característica es un UUID único y estandarizado (cuyo valor es 0x2803) que se utiliza exclusivamente para señalizar la dirección de comienzo de las características. Debido a que los clientes solo pueden recuperar el valor del atributo para conocer el comienzo de las características, solo dispone de permiso de lectura y no de escritura.

Name	Length in bytes	Description
Characteristic Properties	1	A bitfield listing the permitted operations on this characteristic
Characteristic Value Handle	2	The handle of the attribute containing the characteristic value
Characteristic UUID	2, 4, or 16	The UUID for this particular characteristic

Figura 59. Valor de la declaración de característica

El identificador del atributo que contiene el valor actual de la característica está representado en el campo *characteristic value handle*. El campo *characteristic UUID* representa el UUID de la característica y el campo *characteristic properties* indica las operaciones disponibles con la característica.

Property	Location	Description
Broadcast	Properties	If set, allows this characteristic value to be placed in advertising packets, using the Service Data AD Type (see "GATT Attribute Data in Advertising Packets")
Read	Properties	If set, allows clients to read this characteristic using any of the ATT read operations listed in "ATT operations"
Write without response	Properties	If set, allows clients to use the Write Command ATT operation on this characteristic (see "ATT operations")
Write	Properties	If set, allows clients to use the Write Request/Response ATT operation on this characteristic (see "ATT operations")
Notify	Properties	If set, allows the server to use the Handle Value Notification ATT operation on this characteristic (see "ATT operations")
Indicate	Properties	If set, allows the server to use the Handle Value Indication/Confirmation ATT operation on this characteristic (see "ATT operations")
Signed Write Command	Properties	If set, allows clients to use the Signed Write Command ATT operation on this characteristic (see "ATT operations")
Queued Write	Extended Properties	If set, allows clients to use the Queued Writes ATT operations on this characteristic (see "ATT operations")
Writable Auxiliaries	Extended Properties	If set, a client can write to the descriptor described in "Characteristic User Description Descriptor"

Figura 60. Propiedades de una característica

El dispositivo cliente será el encargado de leer estas propiedades para determinar qué operaciones pueden realizarse sobre una característica. Esto es especialmente importante para las propiedades *notify* e *indicate,* ya que estas operaciones son iniciadas por el dispositivo servidor, pero requieren de su habilitación por para del dispositivo cliente mediante el descriptor CCCD (*Client Characteristic Configuration Descriptor).*

Finalmente, el atributo *characteristic value* de una característica contiene los datos que el cliente puede leer o escribir para el intercambio de información. El tipo de este atributo

es el mismo UUID especificado en el campo *characteristic uuid* de la declaración de característica.

Descriptores

Los descriptores son utilizados para proporcionar al cliente metadatos acerca de una característica y su valor. Estos están especificados dentro de la definición de característica y después del atributo *characteristic value.* Estos descriptores están compuestos por un solo atributo, la declaración de descriptor de característica, cuyo UUID es el tipo de descriptor y cuyo valor contiene todo lo que define ese tipo de descriptor particular.

El descriptor más utilizado definido por el GATT es el descriptor CCCD (*Client Characteristic Configuration Descriptor).* El funcionamiento de la mayoría de perfiles depende de este descriptor, con un comportamiento asemejado al de un interruptor, habilitará o deshabilitará las actualizaciones iniciadas por el servidor para la característica a la que se encuentra asociado.

4.2 Objetivos de la implementación

El objetivo que se persigue es la implementación del marco *firmware* utilizado en la programación del dispositivo *RedBear Duo*, que en la plataforma HW/SW actuará como dispositivo *Peripheral* BLE. Este marco servirá como punto de partida para crear una serie de funciones que posibiliten la interacción con un dispositivo *Central* BLE, que en este caso será un terminal móvil, con la centralita de un vehículo. Esta comunicación permitirá al usuario realizar ciertas acciones:

- Conocer qué direcciones de PID tiene el vehículo disponible para la consulta, es decir, después de realizar la petición de la información de disponibilidad desde el dispositivo móvil mediante BLE, el sistema será capaz de enviar al terminal móvil la información solicitada.
- El usuario podrá realizar la petición de una o varias direcciones PID disponibles en el vehículo, y el sistema deberá devolver la información recibida por el vehículo al terminal móvil mediante transmisión BLE.
- El sistema dispondrá de una función que active una periodicidad de envío, lo que implica que el sistema será el encargado de realizar peticiones periódicas hacia los

PID que el usuario haya seleccionado. Esta función supone un refresco automático de la información solicitada por el usuario.

4.3 Diagrama de flujo

El diagrama de flujo asociado con el funcionamiento de la plataforma HW/SW final que se implementará en el presente TFG, es el que se muestra en la Figura 61.

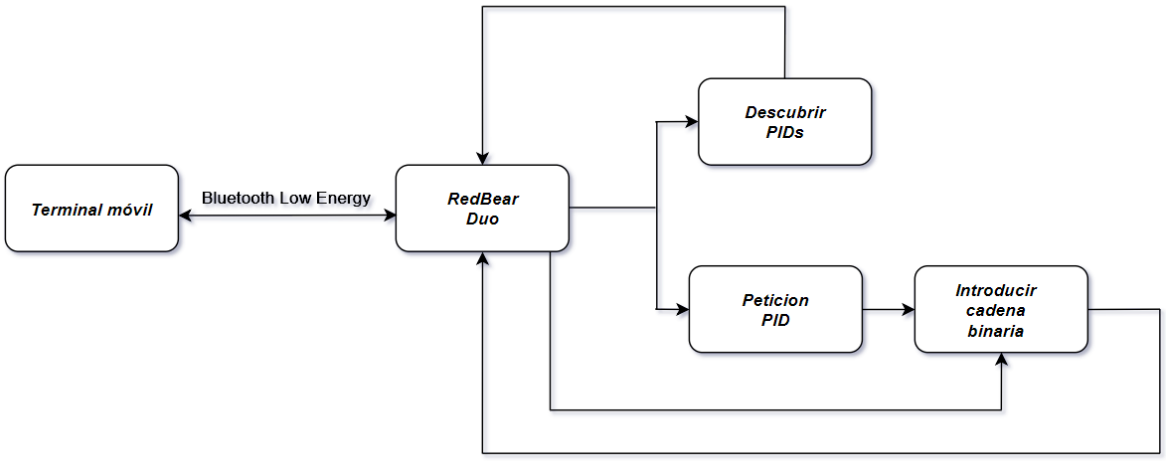

Figura 61. Diagrama de flujo plataforma final HW/SW

Como se indica en la Figura 61, el sistema final se basa en la comunicación del terminal móvil con el dispositivo IoT *RedBear Duo*. Desde el terminal móvil el usuario puede realizar envíos de comandos vía BLE desde el terminal móvil. Si el usuario desea conocer la disponibilidad de PID del vehículo procederá lanzar un comando hacia el dispositivo *RedBear Duo*, éste ejecutará la función principal encargada de interrogar al vehículo. Una vez realizada esta transmisión, el dispositivo *RedBear Duo* enviará la información de disponibilidad al terminal móvil del usuario.

Si, por el contrario, el usuario quiere conocer la información presente en las direcciones PID disponibles del vehículo, podrá enviar una trama de datos que, caracterizada, activará el sistema que será el encargado de realizar las peticiones de información a las direcciones PID del vehículo. Una vez recibida la información el dispositivo *RedBear Duo* enviará la información de vuelta al terminal móvil.

4.4 Desarrollo del *firmware*

Para el desarrollo del *firmware* del dispositivo *RedBear Duo* se ha partido del modelo de referencia proporcionado en el repositorio *Github* de la empresa *RedBear*. En dicho modelo, denominado `SimpleBLEPeripheral.ino`, se establecen las bases que permite a un dispositivo IoT, utilizando el rol de *Peripheral*, establecer una comunicación estable con otro dispositivo que utilice el rol de *Central*. La metodología que se seguirá parte del análisis de las declaraciones de variables y constantes. Una vez analizado esto, se procederá a analizar las funciones implementadas en el código para finalmente, analizar las funciones *"setup()"* y *"loop()"* que determinarán la funcionalidad del sistema.

En la Figura 62 se muestra el primer fragmento correspondiente al código implementado en el dispositivo *RedBear Duo.* En él, se introduce la librería con el objeto *"carloop"* y se definen constantes que más adelante se utilizarán para caracterizar la comunicación BLE.

```
1. #include <carloop.h>
2. Carloop<CarloopRevision2> carloop;
3.
4. #define MIN_CONN_INTERVAL          0x0028 // 50ms.
5. #define MAX_CONN_INTERVAL          0x0190 // 500ms.
6. #define SLAVE_LATENCY              0x0000 // No slave latency.
7. #define CONN_SUPERVISION_TIMEOUT   0x03E8 // 10s.
8.
9. // Learn about appearance:
   http://developer.bluetooth.org/gatt/characteristics/Pages/Character
   isticViewer.aspx?u=org.bluetooth.characteristic.gap.appearance.xml
10.      #define BLE_PERIPHERAL_APPEARANCE  BLE_APPEARANCE_UNKNOWN
11.
12.      #define BLE_DEVICE_NAME            "BLE_Peripheral"
13.
14.      // Length of characteristic value.
15.      #define CHARACTERISTIC1_MAX_LEN    20
16.      #define CHARACTERISTIC2_MAX_LEN    20
17.      #define SAMPLE_PERIOD 4000 // (2s)
18.      #define NUMBER_AVAILABLE_PIDS 32
19.      #define PID_LEN       16 // Discovered PID bitmap - 16 bytes
   = 128 bits
20.      #define N_PID         3  // Number of PID groups - 4
21.      #define N_PID_BITMAP  4  // PID bitmap per PID group - 4
   bytes = 32 bits
22.
23.      // SparkInterval timer
24.      #define SENDPID_TIMING 2000
25.      IntervalTimer sendPidTimer;
26.
```

Figura 62. Definiciones iniciales

Se han definido los intervalos máximos y mínimos de tiempo en la conexión, la latencia admisible en la respuesta por parte del dispositivo *Slave* y el *timeout* establecido para la conexión, así como la definición de aspectos característicos del dispositivo que implementa BLE y la creación de un *timer*. Por motivos de practicidad, las definiciones serán analizadas con mayor profundidad en la sección del código que sean utilizadas.

A continuación, se procederá a analizar las declaraciones y definiciones de las variables que se utilizarán en el fragmento de código mostrado en la Figura 63.

```
1. /*****************************************************
2.  *               Variable Definitions
3.  *****************************************************/
4. // Primary service 128-bits UUID
5. static uint8_t service1_uuid[16] = { 0x71,0x3d,0x00,0x00,0x50,0x3e,
   0x4c,0x75,0xba,0x94,0x31,0x48,0xf1,0x8d,0x94,0x1e };
6. // Characteristics 128-bits UUID
7. static uint8_t char1_uuid[16]
    = { 0x71,0x3d,0x00,0x02,0x50,0x3e,0x4c,0x75,0xba,0x94,0x31,0x48,0x
   f1,0x8d,0x94,0x1e };
8. static uint8_t char2_uuid[16]
    = { 0x71,0x3d,0x00,0x03,0x50,0x3e,0x4c,0x75,0xba,0x94,0x31,0x48,0x
   f1,0x8d,0x94,0x1e };
9.
10.      // GAP and GATT characteristics value
11.      static uint8_t  appearance[2] = {
12.        LOW_BYTE(BLE_PERIPHERAL_APPEARANCE),
13.        HIGH_BYTE(BLE_PERIPHERAL_APPEARANCE)
14.      };
15.
16.      static uint8_t  change[4] = {
17.        0x00, 0x00, 0xFF, 0xFF
18.      };
19.
20.      static uint8_t  conn_param[8] = {
21.        LOW_BYTE(MIN_CONN_INTERVAL), HIGH_BYTE(MIN_CONN_INTERVAL),
22.        LOW_BYTE(MAX_CONN_INTERVAL), HIGH_BYTE(MAX_CONN_INTERVAL),
23.        LOW_BYTE(SLAVE_LATENCY), HIGH_BYTE(SLAVE_LATENCY),
24.
   LOW_BYTE(CONN_SUPERVISION_TIMEOUT), HIGH_BYTE(CONN_SUPERVISION_TIME
   OUT)
25.      };
26.
```

Figura 63. Variables del firmware del dispositivo RedBear Duo

En el fragmento representado en la Figura 63 se aprecia la definición del UUID del servicio primario, junto con la definición del UUID de las dos características asociadas a la información que se transferirá en la comunicación BLE.

Además, en este fragmento de código se caracteriza el valor de característica de los perfiles GAP y GATT, así como la creación de parámetros que serán posteriormente utilizados para establecer la conexión BLE.

En el fragmento de código indicado en la Figura 64 se aprecia la inicialización de los parámetros propios y del vector de datos correspondientes a los paquetes de *Advertising.*

```
1. static advParams_t adv_params = {
2.   .adv_int_min   = 0x0030,
3.   .adv_int_max   = 0x0030,
4.   .adv_type      = BLE_GAP_ADV_TYPE_ADV_IND,
5.   .dir_addr_type = BLE_GAP_ADDR_TYPE_PUBLIC,
6.   .dir_addr      = {0,0,0,0,0,0},
7.   .channel_map   = BLE_GAP_ADV_CHANNEL_MAP_ALL,
8.   .filter_policy = BLE_GAP_ADV_FP_ANY
9. };
10.
11.      // BLE peripheral advertising data
12.      static uint8_t adv_data[] = {
13.        0x02,
14.        BLE_GAP_AD_TYPE_FLAGS,
15.        BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE,
16.
17.        0x11,
18.        BLE_GAP_AD_TYPE_128BIT_SERVICE_UUID_COMPLETE,
19.        0x1e, 0x94, 0x8d, 0xf1, 0x48, 0x31, 0x94, 0xba, 0x75, 0x4c,
    0x3e, 0x50, 0x00, 0x00, 0x3d, 0x71
20.      };
21.
22.      // BLE peripheral scan respond data
23.      static uint8_t scan_response[] = {
24.        0x08,
25.        BLE_GAP_AD_TYPE_COMPLETE_LOCAL_NAME,
26.        'C', 'A', 'R', 'L', 'O', 'O', 'P'
27.      };
28.
29.      // Characteristic value handle
30.      static uint16_t character1_handle = 0x0000;
31.      static uint16_t character2_handle = 0x0000;
32.      // Buffer of characterisitc value.
33.      uint8_t characteristic1_data[CHARACTERISTIC1_MAX_LEN] = {0x00
   };
34.      uint8_t characteristic2_data[CHARACTERISTIC2_MAX_LEN] = {0x00
   };
35.
36.      uint16_t characteristic1_data_len = CHARACTERISTIC1_MAX_LEN;
37.      uint16_t characteristic2_data_len = CHARACTERISTIC2_MAX_LEN;
```

Figura 64.Variables del firmware del dispositivo RedBear Duo

Se ha establecido la información relativa al nombre del dispositivo que actúa como *Peripheral*, es decir, del *RedBear Duo*, a "CARLOOP" con el objetivo de identificar más fácilmente el dispositivo.

Una vez definidos los parámetros de conexión, se caracterizan los vectores que contendrán la información que se almacenará en las características *char1* y *char2*.

```
1. int flag = 1;
2. bool respuesta = false;
3.
4. //direcciones cadenas de PIDs
5. const auto Disp_PIDS_01_20 = 0x00;
6. const auto Disp_PIDS_21_40 = 0x20;
7. const auto Disp_PIDS_41_60 = 0x40;
8. const auto Disp_PIDS_61_80 = 0x60;
9. //Direcciones PIDs
10.
11.      // IDs de los mensajes OBD
12.      const auto OBD_REQUEST_ID      = 0x7E0;
13.      const auto OBD_REPLY_ID        = 0x7E8;
14.      const auto ID_broadcast      = 0x7DF;
15.      const auto ID_respuesta_minima = 0x7E8;
16.      const auto ID_respuesta_maxima = 0x7EF;
17.
18.
19.      //numero de pids a preguntar/ sin limitacion por ello uso
   size_t
20.      const size_t numero_pids = 3;
21.
22.      const uint8 t pidsToRequest[numero pids] = {
23.
24.      Disp_PIDS_01_20,
25.      Disp_PIDS_21_40,
26.      Disp_PIDS_41_60,
27.      //Disp_PIDS_61_80
28.
29.      };
30.
31.
32.      // modo de operacion
33.      const auto OBD_PID_SERVICE     = 0x01;
34.
```

Figura 65. Variables del firmware del dispositivo RedBear Duo

En el fragmento de código de la Figura 65 se indican una serie de variables adicionales incluidas en el código del *firmware.* En esta sección se definen las variables *"flag"*, que será utilizada como señal interna de funcionamiento, y *"respuesta"*, que se utiliza como señalizador de una recepción correcta.

Además de estas variables, se incluyen las direcciones de los cuatro grupos de PID de disponibilidad a los que se realizará una petición con el fin de determinar qué PID están disponibles. También se han incluido las ID correspondientes a las ID que se utilizarán en el campo ID de la estructura *CANMessage*, que se analizó en la implementación de las funciones básicas en el dispositivo *Photon*.

Con el objetivo de gestionar de manera más práctica las direcciones de los PID de disponibilidad, se generó un vector que las almacena. La dirección correspondiente al cuarto grupo de PID *(Disp_PIDS_61_80)* se ha eliminado, ya que el vehículo de pruebas solo dispone de los tres primeros grupos de PID.

Por último, se define el modo de operación que se utilizará para las peticiones que se realicen al vehículo *(Modo 1).*

El resto de las variables serán analizadas dentro de las funciones que las utilicen a fin de realizar un análisis del código óptimo.

4.4.1 Funciones de comunicación

En esta sección se procederá a analizar las funciones propias de la implementación BLE, que incluyen aquellas que permiten establecer una correcta comunicación entre dispositivos. En la Figura 66 se indica el fragmento de código correspondiente a la función *"deviceConnectedCallback()".*

```
1. uint16_t connected_id = 0xFFFF;
2. void deviceConnectedCallback(BLEStatus_t status, uint16_t handle) {
3.   switch (status) {
4.     case BLE_STATUS_OK:
5.       Serial.println("Device connected!");
6.       connected_id = handle;
7.       Serial.print("  - Device connected handle: ");
8.       Serial.println(connected_id);
9.       break;
10.          default: break;
11.        }
12.      }
13.
```

Figura 66. DeviceConnectedCallback

En dicha función se determina el estado de la conexión BLE y se muestran por el terminal serie un conjunto de parámetros básicos de conexión (*Handler* y la ID correspondiente).

```
1. void deviceDisconnectedCallback(uint16_t handle) {
2.   Serial.println("Device disconnected!");
3.   connected_id = 0xFFFF;
4. }
```

Figura 67. Función deviceDisconnectedCallback()

La función indicada en la Figura 67 hace referencia, de manera análoga a la función descrita en la Figura 66, a un mensaje que se emite en caso de desconectarse el dispositivo.

```
1. uint16_t gattReadCallback(uint16_t value_handle, uint8_t * buffer,
   uint16_t buffer_size) {
2.   uint8_t characteristic_len = 0;
3.
4.   Serial.print("Read value handler: ");
5.   Serial.println(value_handle, HEX);
6.
7.   if (character1_handle == value_handle) {
8.     Serial.print("  - characteristic1 read: ");
9.     memcpy(buffer, characteristic1_data, characteristic1_data_len);
10.          characteristic_len = CHARACTERISTIC1_MAX_LEN;
11.
     for (uint8_t index = 0; index < CHARACTERISTIC1_MAX_LEN; index++)
    {
12.            Serial.print(buffer[index], HEX);
13.          }
14.          Serial.println();
15.        }
16.        else if (character2_handle == value_handle) {
17.          Serial.print("  - characteristic2 read: ");
18.
     memcpy(buffer, characteristic2_data, CHARACTERISTIC2_MAX_LEN);
19.          characteristic_len = CHARACTERISTIC2_MAX_LEN;
20.
     for (uint8_t index = 0; index < CHARACTERISTIC2_MAX_LEN; index++)
    {
21.            Serial.print(buffer[index], HEX);
22.          }
23.          Serial.println();
24.        }
25.        return characteristic_len;
26.      }
```

Figura 68. Función gattReadCallback()

El fragmento de código que aparece en la Figura 68 corresponde a la función *"gattReadCallback()"* que realiza la lectura implementada por el perfil GATT. Esta función representa vía serie la información que contienen las dos características definidas ante una petición de lectura, además de devolver en formato *uint16_t*, la longitud de la característica leída.

```
1. bool new_data = false;
2. int gattWriteCallback(uint16_t value_handle, uint8_t *buffer, uint1
   6_t size) {
3.   Serial.print("Write value handler: ");
4.   Serial.println(value_handle, HEX);
5.
6.   if (character1_handle == value_handle) {
7.     characteristic1_data_len = size;
```

```
8.      memcpy(characteristic1_data, buffer, size);
9.      Serial.print("  - characteristic1 write value len: ");
10.           Serial.println(characteristic1_data_len);
11.           Serial.print("  - characteristic1 write value: ");
12.           for (uint8_t index = 0; index < size; index++) {
13.             Serial.print(characteristic1_data[index], HEX);
14.             Serial.print(" ");
15.           }
16.           Serial.println(" ");
17.           new_data = true;
18.         }
19.         else if (character2_handle+1 == value_handle) {    // Client
    Characteristic Configuration Descriptor Handle.
20.           Serial.print("  - characteristic2 CCCD write value: ");
21.           for (uint8_t index = 0; index < size; index++) {
22.             Serial.print(buffer[index], HEX);
23.           }
24.           Serial.println(" ");
25.         }
26.         return 0;
27.       }
28.
29.
```

Figura 69. Función gattWriteCallback()

La función *"gattWriteCallback()",* indicada en el fragmento de código de la Figura 69, realiza la función de escritura análoga a la función de lectura mostrada en la Figura 68.

4.4.2 Funciones de procesamiento de datos

En esta sección se analizarán las funciones que se han desarrollado específicamente para la implementación del funcionamiento básico de la plataforma *HW/SW* final.

```
1. void DescubrirPIDS() {
2.      respuesta = false;
3.
4.      if ( grupo_pids == 3){
5.          pids_descubiertos = true;
6.          grupo_pids = 0;
7.
8.          return;
9.      }
10.
11.
12.           while(respuesta == false){
13.
14.           CANMessage mensaje;
15.           mensaje.id = OBD_REQUEST_ID; //Id del tipo de mensaje
16.           mensaje.len = 8; //longitud del mensaje (caracteristica
    del objeto message)
17.           mensaje.data[0] = 0x02;
18.           mensaje.data[1] = OBD_PID_SERVICE;
```

```
19.           mensaje.data[2] = pidsToRequest[grupo_pids];
20.           carloop.can().transmit(mensaje);
21.           delay(600);
22.           respuesta = false;
23.
24.
25.          Serial.println(pidsToRequest[grupo_pids]);
26.
27.           CANMessage mensaje2;
28.           while(carloop.can().receive(mensaje2)) {
29.
     if(mensaje2.id == OBD_REPLY_ID && mensaje2.data[2] == pidsToReque
   st[grupo_pids]) {
30.                     respuesta = true;
31.                     pid_bitmap[grupo_pids][0] = mensaje2.data[3];
32.                     pid_bitmap[grupo_pids][1] = mensaje2.data[4];
33.                     pid_bitmap[grupo_pids][2] = mensaje2.data[5];
34.                     pid_bitmap[grupo_pids][3] = mensaje2.data[6];
35.
36.                     }
37.
38.               }
39.          }
40.               grupo_pids++;
41.               DescubrirPIDS();
42.
43.           }
```

Figura 70. Función DescubrirPIDS()

La función *"DescubrirPIDS()"* se basa en la analizada en la sección 3.3.2 "Disponibilidad de PIDs y peticiones de información". Para la función, en su forma final, se ha incluido un sistema reentrante que proporcione las direcciones de los PID disponibles.

La función realizará peticiones individuales a cada dirección correspondiente a los PID de disponibilidad. Después de cada petición el sistema quedará a la espera de la respuesta por parte del vehículo a la primera dirección de disponibilidad (0x00). Cuando haya recibido la respuesta al primer grupo de disponibilidad, la función volverá a ejecutarse con la siguiente dirección del PID de disponibilidad (0x20).

Una vez se haya completado el vector *"pid_bitmap[grupo_pids][]"*, donde el primer argumento corresponde al grupo de disponibilidad (del primero hasta el tercero), y el segundo argumento hace referencia al índice del vector en el que se almacenará la información de disponibilidad que devuelva el vehículo, el *loop* principal del sistema continuará la ruta de ejecución encargada de enviar la información vía BLE, que se analizará más adelante.

```
1. String hallar_pids(uint8_t pid_recieved) {
```

```
2.
3.      uint8_t filtro = 0x01;
4.      uint8_t filtro2 = 0x00;
5.      pid_procesado = "";
6.
7.      for(int i = 0; i < 8 ; i++)
8.      {
9.          filtro = 0x01;
10.             filtro2 = filtro << i;
11.
12.             if((filtro2&pid_recieved))
13.                 pid_procesado= "1" + pid_procesado;
14.             else{
15.                 pid_procesado= "0" + pid_procesado;
16.             }
17.
18.
19.
20.         }
21.
22.         return pid_procesado;
23.     }
```

Figura 71. Función hallar_pids()

La función *"hallar_pids()"*, mostrada en la Figura 71, se ha creado con el objetivo de representar vía puerto serie la información de disponibilidad recibida desde el vehículo. Esta información será recibida en forma de *bytes* y resulta práctico observar la cadena binaria en el ordenador portátil para contrastar la información recibida, con la que se recibió en el apartado 3.3.2 "Disponibilidad de PID y peticiones de información".

```
1. void PeticionPID(int grupo , int PID) {
2.      //limpiamos array de trama BLE
3.
4.      for(uint8_t n = 0; n <20 ; n++){
5.            characteristic2_data[n] = 0x0;
6.          }
7.      //comprobacion pids disponibles
8.      //bool respuesta = false;
9.      if (pids_descubiertos == true){
10.           bin32 ="";
11.
12.           switch(grupo){
13.               case 0:
14.               PID = PID + 0;
15.               break;
16.               case 1:
17.               PID = PID + 32;
18.               break;
19.               case 2:
20.               PID = PID + 64;
21.               break;
22.               case 3:
23.               PID = PID + 128;
24.               break;
25.
```

```
26.           }
27.
28.
29.           CANMessage mensaje;
30.           mensaje.id = ID_broadcast; //Id del tipo de mensaje
31.           mensaje.len = 8; //longitud del mensaje (caracteristica
   del objeto message)
32.           mensaje.data[0] = 0x02;
33.           mensaje.data[1] = OBD_PID_SERVICE;
34.           mensaje.data[2] = PID;
35.           carloop.can().transmit(mensaje);
36.           delay(600);
37.
38.           CANMessage mensaje2;
39.           while(carloop.can().receive(mensaje2)) {
40.
     if (mensaje2.id >= ID_respuesta_minima && mensaje2.id <= ID_respu
   esta_maxima && mensaje2.data[2] == PID) {
41.                   respuesta = true;
42.                   dato0 = mensaje2.data[0];
43.                   dato1 = mensaje2.data[1];
44.                   //PID code
45.                   dato2 = mensaje2.data[2];
46.                   //datos
47.                   dato3 = mensaje2.data[3];
48.                   dato4 = mensaje2.data[4];
49.                   dato5 = mensaje2.data[5];
50.                   dato6 = mensaje2.data[6];
51.                   //no tiene informacion de interes
52.                   dato7 = mensaje2.data[7];
53.
54.                   RecibirPID(PID, dato3, dato4 ,dato5 , dato6);
55.               }
56.           }
57.
58.       }else{
59.           Serial.println("Es necesario Descubrir los pids
   disponibles primero..");
60.       }
61.       }
```

Figura 72. Función PeticiónPID()

La función *"PeticionPID(grupo)(PID)"* es la encargada de realizar las peticiones al vehículo, recibiendo como parámetros el grupo de PID al que se hace referencia (1-3), y la dirección o PID de la que se desea recibir la información.

En primera instancia, el sistema inicializará los valores almacenados en el vector de datos correspondiente a la característica en la que se almacena la información recibida desde el vehículo *(characteristic2_data)*. Una vez inicializado el vector, es necesario comprobar que el usuario haya realizado previamente un descubrimiento previo de los PID disponibles.

Asumiendo que el usuario ya ha ejecutado la función encargada de descubrir los PID, se valora en qué grupo de PID se encuentra la dirección del PID objetivo, esto es, en caso de que el usuario haya especificado el segundo grupo de PID, el sistema sumará automáticamente los 32 valores previos de PID, obteniendo así el índice correcto cuando se analice el *bitmap* introducido por el usuario.

Después de ajustar la dirección PID a la dirección correcta, se procede a realizar la petición de información, y posterior recepción de la información proveniente del vehículo. Esta información será almacenada en las variables destinadas para ello, de tipo *uint8_t*, como se observa en la Figura 73.

```
1. uint8_t dato0;
2. uint8_t dato1;
3. uint8_t dato2;
4. uint8_t dato3;
5. uint8_t dato4;
6. uint8_t dato5;
7. uint8_t dato6;
8. uint8_t dato7;
```

Figura 73. Variables para almacenar la información proveniente del vehículo (máximo 8 bytes)

Una vez almacenada la información, se ejecutará la función *"RecibirPID()"*, cuya implementación se muestra en el fragmento de código de la Figura 74.

```
1. void RecibirPID(int pid, uint8_t dato3, uint8_t dato4, uint8_t dato
   5, uint8_t dato6) {
2.
3.     switch( pid ){
4.   case 0:
5.   case 32:
6.   case 64:
7.   case 96:{
8.   }
9.             break;
10.        //4 bytes
11.        case 1:
12.        case 36:
13.        case 37:
14.        case 38:
15.        case 39:
16.        case 40:
17.        case 41:
18.        case 42:
19.        case 43:
20.        case 52:
21.        case 53:
22.        case 54:
23.        case 55:
24.        case 56:
25.        case 57:
```

```
26.         case 58:
27.         case 59:
28.         case 65:
29.         case 79:{
30.                     dato_util[indice_dato_util] = dato3;
31.                     dato_util[indice_dato_util + 1] = dato4;
32.                     dato_util[indice_dato_util + 2] = dato5;
33.                     dato_util[indice_dato_util + 3] = dato6;
34.                     indice_dato_util = indice_dato_util + 4;
35.                     llenado_vector = llenado_vector + 4;
36.                 }
37.                 break;
38.         //2 bytes
39.
40.         case 2:
41.         case 3:
42.         //case 12:
43.         case 16:
44.         case 20:
45.         case 21:
46.         case 22:
47.         case 23:
48.         case 24:
49.         case 25:
50.         case 26:
51.         case 27:
52.         case 31:
53.         case 33:
54.         case 34:
55.         case 35:
56.         case 49:
57.         case 50:
58.         case 60:
59.         case 61:
60.         case 62:
61.         case 63:
62.         case 66:
63.         case 67:
64.         case 68:
65.         case 77:
66.         case 78:
67.         case 83:
68.         case 84:
69.         case 85:
70.         case 86:
71.         case 87:
72.         case 88:
73.         case 89:
74.         case 93:
75.         case 94:
76.         case 99:
77.         case 101:{
78.                     dato_util[indice_dato_util] = dato3;
79.                     dato_util[indice_dato_util + 1] = dato4;
80.                     indice_dato_util = indice_dato_util + 2;
81.                      llenado_vector = llenado_vector + 2;
82.         }
83.         break;
84.
85.         //1 byte
86.         case 4:
```

```
87.         case 5:
88.         case 6:
89.         case 7:
90.         case 8:
91.         case 9:
92.         case 10:
93.         case 11:
94.         case 13:
95.         case 14:
96.         case 15:
97.         case 17:
98.         case 18:
99.         case 19:
100.        case 28:
101.        case 29:
102.        case 30:
103.        case 44:
104.        case 45:
105.        case 46:
106.        case 47:
107.        case 48:
108.        case 51:
109.        case 69:
110.        case 70:
111.        case 71:
112.        case 72:
113.        case 73:
114.        case 74:
115.        case 75:
116.        case 76:
117.        case 81:
118.        case 82:
119.        case 90:
120.        case 91:
121.        case 92:
122.        case 95:
123.        case 97:
124.        case 98:
125.        case 125:
126.        case 126:{
127.                    dato_util[indice_dato_util] = dato3;
128.                    indice_dato_util = indice_dato_util + 1;
129.                     llenado_vector = llenado_vector + 1;
130.        }
131.        break;
132.
133.        case 12: {
134.                     dato util[indice dato util] = dato3;
135.                    dato_util[indice_dato_util + 1] = dato4;
136.                    indice_dato_util = indice_dato_util + 2;
137.                     llenado_vector = llenado_vector + 2;
138.
139.                    Serial.println("Revoluciones del motor");
140.                    uint8_t engine_rpmh = ((256*dato3)+dato4)/4;
141.                    float engine_rpm = ((256*dato3)+dato4)/4;
142.                    Serial.println(engine_rpm);
143.                    Serial.print("rpm");
144.                    Serial.println("en hexagesimal dato 3");
145.                    Serial.println(dato3, HEX);
146.                    Serial.println("en hexagesimal dato 4");
147.                    Serial.println(dato4, HEX);
```

```
148.
149.          }
150.          break;
151.      }
152.      }
```

Figura 74. Función RecibirPID()

La función *"RecibirPID()"* es de especial importancia, debido a que en vistas a rellenar el vector de datos de la característica en la que se almacenará la información que devuelva el vehículo, como se ha analizado en el apartado 4.1.3 "Características básicas del protocolo", el tamaño máximo de este vector es de 20 *bytes*.

En el apartado 2.2.3 "Direcciones de PID estandarizados" se indicó la cantidad de *bytes* de respuesta que corresponde a cada dirección PID del vehículo. En consecuencia, el sistema es el encargado de ir almacenando en orden los datos en el vector *"dato_util[]"*, que será el que finalmente rellene el vector de datos de la característica. Para ello, se ha codificado una función *switch* que será la encargada de rellenar ese vector de información y actualizar los índices, teniendo en cuenta la cantidad de *bytes* de respuesta que corresponde a la dirección PID objetivo.

Durante la codificación se realizó el procesado de un PID al que poder acceder de manera simple (revoluciones del motor, dirección 0x0C) y que permitiera una posterior verificación más sencilla del código.

En el fragmento de código mostrado en la Figura 75 se indica la codificación de la función *"EnviarBytes()"*, que será la encargada de realizar los ajustes finales en el proceso de rellenar el vector de datos de la característica, respetando el tamaño máximo de trama de datos transmitida por BLE.

```
1. void EnviarBytes(int indice[]) {
2.
3.
4.     uint8_t contador_aux = contador_rellenar;
5.
     for(uint8_t indice_rellenar = 0; indice_rellenar < CHARACTERISTIC
   2_MAX_LEN; indice_rellenar++ ){
6.
     if(indice_rellenar + PID_bytes[(indice[contador_aux]+1)] >= 20 ){
7.             Serial.println("Se ha enviado ya una trama");
8.
9.
   ble.sendNotify(character2_handle, characteristic2_data, CHARACTERIS
   TIC2_MAX_LEN);
10.
```

```
11.                 //envio trama anterior y limpio
12.                 //relleno siguiente y actualizo indice
13.                 for(uint8_t p = 0; p <20 ; p++){
14.                 characteristic2_data[p] = 0x0;
15.               }
16.               indice_rellenar = 0;
17.               }
18.
19.
     uint8_t tamano = PID_bytes[(indice[contador_aux]+1)];
20.                 Serial.println("tamano: ");
21.
   Serial.println(PID_bytes[(indice[contador_aux]+1)]);
22.                 switch(tamano){
23.                     case 0:
24.                       break;
25.                     case 1:
26.
   characteristic2_data[indice_rellenar] = dato_util[contador_rellenar
   ];
27.
28.                       contador_rellenar = contador_rellenar + 1;
29.                       contador_aux++;
30.                       break;
31.                     case 2:
32.
   characteristic2_data[indice_rellenar] = dato_util[contador_rellenar
   ];
33.
   characteristic2_data[indice_rellenar + 1] = dato_util[contador_rell
   enar + 1];
34.                       indice_rellenar = indice_rellenar + 1;
35.                       contador_rellenar = contador_rellenar + 2;
36.                       contador_aux++;
37.                       break;
38.                     case 4:
39.
   characteristic2_data[indice_rellenar] = dato_util[contador_rellenar
   ];
40.
   characteristic2_data[indice_rellenar + 1] = dato_util[contador_rell
   enar + 1];
41.
   characteristic2_data[indice_rellenar + 2] = dato_util[contador_rell
   enar + 2];
42.
   characteristic2_data[indice_rellenar + 3] = dato_util[contador_rell
   enar + 3];
43.                       indice_rellenar = indice_rellenar +  3;
44.                       contador_rellenar = contador_rellenar + 4;
45.                       contador_aux++;
46.
47.                       break;
48.                 }
49.                 Serial.println("valores de contador rellenar y
   llenado vector respectivamente");
50.                 Serial.println(contador_rellenar, DEC);
51.                 Serial.println(llenado_vector, DEC);
52.                 //contador_rellenar++;
53.                 if(contador_rellenar == (llenado_vector)){
54.
```

```
55.                     contador_rellenar = 0;
56.                     llenado_vector = 0;
57.
58.
    ble.sendNotify(character2_handle, characteristic2_data, CHARACTERI
   STIC2_MAX_LEN);
59.              //Particle.process();
60.              return;
61.           }
62.             }
63.
64.             contador_rellenar =0;
65.             return;
66.         }
```

Figura 75. Función EnviarBytes()

Esta función, en primera instancia, comprueba que no se encuentre lleno el vector de datos de la característica. En caso de haberse llenado, el primer condicional forzará al envío de la trama de datos, ya que se limita a una longitud de 20 *bytes*. Una vez que se ha enviado la trama mediante una notificación, se puede proceder a rellenarla de nuevo.

Después de haber comprobado que el vector de datos no se encuentra lleno, el sistema analiza el tamaño que tener el vector de datos final, es decir, se suma el tamaño en *bytes* correspondiente a las peticiones específicas que haya realizado el usuario. Esto sirve para rellenar el vector de forma ordenada y capacita al sistema para identificar cuándo la trama de datos está completa, enviarla, y comenzar a rellenar la siguiente.

Para el cálculo del tamaño del vector final se ha definido un vector de carácter estático que incluye toda la información relativa a la cantidad de *bytes* de información útil que tiene cada PID, mostrado en la Figura 76. Así, en orden, el sistema puede averiguar cuántos *bytes* tiene que rellenar en total en el vector de datos de la característica.

```
static uint8_t PID_bytes[100] = {0,4,2,2,1,1,1,1,1,1,1,1,2,1,1,1,2,1,1,1,
2,2,2,2,2,2,2,2,1,1,1,2,4,2,2,2,4,4,4,4,4,4,4,4,1,1,1,1,1,2,2,1,4,4,4,4,4
,4,4,4,2,2,2,2,0,4,2,2,2,1,1,1,1,1,1,1,1,2,2,4,4,1,1,2,2,2,2,2,2,2,1,1,1,
2,2,1,0,1,1,2};
```

Figura 76. Vector de tamaño de bytes útiles

El *firmware* desarrollado para el dispositivo *RedBear Duo* en la plataforma HW/SW final se ha enfocado a primar el ahorro y la eficiencia energética, minimizando las conexiones necesarias para transmitir la información de manera eficiente.

En el fragmento de código mostrado en la Figura 77 se detalla la función *"sendPid()"*, que será llamada periódicamente en caso de que el usuario desee obtener la información que ha solicitado por medio de la introducción del *bitmap*, refrescada automáticamente en el dispositivo.

```
1. void sendPid(void) {
2.
3.   if ((pid_notify_enable) && (ble.attServerCanSendPacket())) {
4.
     for(int indice_contador_timer = 0; indice_contador_timer < contad
   or; indice_contador_timer++ ){
5.
   PeticionPID(pid_global , (indice[indice_contador_timer]+1));
6.
7.    }
8.          EnviarBytes(indice);
9. }
10.      }
```

Figura 77. Función SenPid()

4.4.3 Función loop()

Para analizar el código de manera práctica, se mostrará la función *loop()* principal del firmware correspondiente al dispositivo *RedBear Duo* por fragmentos, correspondiendo cada fragmento de código a la sección asociada al procesamiento de la elección del usuario.

```
1. void loop() {
2.    if (new_data) {
3.      new_data = false;
4.      Serial.print("* COMMAND = ");
5.
    for (uint8_t index = 0; index < characteristic1_data_len; index++)
    {
6.        command[index] = characteristic1_data[index];
7.        Serial.print(command[index], HEX);
8.        Serial.print(" ");
9.      }
10.           Serial.println("");
11.
12.           //Parse command
13.           if ((command[0] == 0xFE) && (command[1] == 0xFF)) {
14.              Serial.println("*** OK - valid command format");
15.              uint8_t option = command[2];
16.
17.
18.              if (option == 0x01) { // 1. Discover PID
19.              Serial.println("Descubrir pids seleccionado:  ");
20.
```

```
21.
    for (uint8_t index = 0; index < CHARACTERISTIC2_MAX_LEN; index++)
   {
22.                     characteristic2_data[index] = 0x00;
23.                 }
24.
25.                     DescubrirPIDS();
26.                 Serial.print("***      Send PID ");
27.                 int p =0;
28.                 int q=0;
29.                for (uint8_t index = 0; index < PID_LEN -
    4; index++) {
30.
31.
32.
   characteristic2_data[index] = pid_bitmap[p][q];
33.
34.                     Serial.print(characteristic2_data[index], BIN);
35.                     Serial.print(" ");
36.
37.                     q++;
38.                     if(q > 4){
39.                         p++;
40.                         q=0;
41.
42.                             }
43.
44.                 }
45.
46.                 Serial.println("");
47.
48.
    ble.sendNotify(character2_handle, characteristic2_data, CHARACTERI
   STIC2_MAX_LEN);
49.             }
```

Figura 78. Función loop opción 1

En la sección de código que aparece en la Figura 78 se procesa el primer caso de uso, en el que el usuario desea conocer qué direcciones PID están disponibles en el vehículo.

Se ha utilizado una nomenclatura específica para los comandos por parte del usuario. Así, los dos primeros *bytes* de la trama se definen como 0xFE y 0xFF, siendo indispensable que estos *bytes* tengan estos valores para que el sistema detecte la trama recibida vía BLE como una trama válida.

Por ejemplo, si el usuario desea conocer la disponibilidad de PID, deberá introducir el comando 0xFEFF01. Mediante este comando el sistema detectará la trama válida y procesará el siguiente *byte* como la opción que el usuario desea ejecutar (en este caso la opción “1”).

Una vez que el sistema detecta la opción requerida, inicializa el vector de datos de la característica, para proceder a ejecutar la función *DescubrirPIDs()* con el fin de obtener los *bitmaps* de disponibilidad. Estos *bitmaps* de disponibilidad serán almacenados en la variable *"pid_bitmap[][]"*, que es utilizada posteriormente para asignar su valor dentro del vector de datos de la característica.

Finalmente, se envía la trama de datos sin necesidad de más comprobaciones, ya que el tamaño máximo que ocupa la información relativa a la disponibilidad de PID es de 16 *bytes* (4 *bytes* por cada *bitmap*, el cual no excede los 20 bytes de datos del vector asociado a la característica.

A continuación, en la Figura 79 se muestra el fragmento de código que procesa la segunda opción de usuario, esto es, obtener la información relativa a los PID que el usuario incluya en el *bitmap* que introduce en el sistema. Esta introducción se realiza desde el terminal móvil, siguiendo en primer lugar la nomenclatura de los dos primeros *bytes* establecidos (0xFE y 0xFF). A continuación, para acceder a la funcionalidad es necesario introducir, además de los dos primeros *bytes* definidos, la opción elegida –"2"-, el grupo de PID al que hace referencia la petición, y el *bitmap* propiamente dicho en el que se incluyen los PID de los que se desea recibir la información.

En consecuencia, si por ejemplo el usuario deseara recibir la información relativa a los cuatro primeros PID del vehículo, la trama de envío desde el terminal móvil sería de la forma 0xFEFF0200F0000000. Esta trama de 8 *bytes* representa, en orden:

- **FEFF:** Formato de inicio especificado.
- **02:** Opción elegida -2-.
- **00:** Grupo de PID elegido (1º grupo de PID).
- **F0000000:** Se requiere conocer los cuatro primeros PID (F0 = 11110000).

```
1. else if (option == 0x02) { // 2. Enter PID to receive info from
2.            Serial.println("***      OPTION = 2");
3.            uint8_t pid = command[3];
4.            pid_global = pid;
5.            Serial.print("***         PID = ");
6.            Serial.println(pid, HEX);
7.
8.
9.            Serial.print("***         PID BITMAP = ");
```

```
10.
    for (uint8_t index = 0; index < N_PID_BITMAP; index++) {
11.                     Serial.print(pid_bitmap[pid][index]);
12.                     Serial.print(" ");
13.                 }
14.                 Serial.println("Numero binario elegido");
15.
16.                  String PIDS = hallar pids(command[4]);
17.                   PIDS = PIDS + hallar_pids(command[5]);
18.                   PIDS = PIDS + hallar_pids(command[6]);
19.                   PIDS = PIDS + hallar_pids(command[7]);;
20.
21.                  Serial.println(PIDS);
22.
23.                  Serial.println("se requieren conocer: ");
24.                  respuesta = false;
25.
26.                  contador = 0;
27.                  for(int h = 0;h<32;h++){
28.
29.
30.                      if (PIDS[h] == '1'){
31.                          Serial.println("indice a buscar");
32.                          Serial.println(h+1);
33.                          indice[contador] = (h);
34.                          contador++;
35.                      }
36.                   }
37.                   Serial.println("valor en primera posicion
   vector");
38.                   Serial.println(indice[0]);
39.
    for(int indice_contador = 0; indice_contador < contador; indice_co
   ntador++ ){
40.
41.
42.                   Serial.print(" , ");
43.                   while(respuesta == false){
44.                        if(n_envios == 4){
45.                          Serial.println("pid no disponible: ");
46.
   Serial.println((indice[indice_contador]+1));
47.                          n_envios = 0;
48.                          respuesta = true;
49.                          Particle.process();
50.                        }
51.                       Serial.println("preguntando por");
52.                       Serial.println((indice[indice contador])+1);
53.
    PeticionPID(pid , ((indice[indice_contador]+1)));
54.                       n_envios++;
55.
56.                      }
57.                      respuesta = false;
58.                   }
59.                   indice_dato_util = 0;
60.
61.                  Serial.println("sali del bucle");
62.
63.                   EnviarBytes(indice);
64.
```

```
65.                     Particle.process();
66.
67.
68.                     }
```

Figura 79. Función loop opción 2

Una vez se ha identificado la trama recibida por parte del usuario, como una trama correcta, se procede a identificar los PID escogidos para las peticiones por medio de la función *"hallar_pids()"*. Mediante esta función se obtiene un vector de índices que será el que caracterice las peticiones por medio de la función *"PeticionPID()"*.

Realizada la ejecución de la función *"PeticionPID()"* a las direcciones proporcionadas, se procede a ejecutar la función *"EnviarBytes()"*, que es la encargada de transmitir la información recibida desde el vehículo, al terminal móvil, vía BLE.

Además de una protección en caso de que se haya seleccionado un PID que no se encuentra disponible, se han utilizado diversas impresiones vía puerto serie con el objetivo de realizar la verificación del código de manera más efectiva, sin necesidad de afectar a la funcionalidad de éste, ya que la trama recibida en el terminal móvil no varía.

```
1. else if (option == 0x03) { // 3. Start receiving info from PID
2.             Serial.println("***      OPTION = 3");
3.             Serial.println("***         Start sending info from
   PID");
4.             pid_notify_enable = true;
5.             sendPidTimer.begin(sendPid, SENDPID_TIMING, hmSec);
6.          }
7.          else if (option == 0x04) { // 4. Stop receiving info from
   PID
8.             Serial.println("***      OPTION = 4");
9.             Serial.println("***         End sending info from PID");
10.                pid_notify_enable = false;
11.                sendPidTimer.end();
12.             }
13.             else {
14.                Serial.print("*** ERROR - OPTION not valid (");
15.                Serial.print(option, HEX);
16.                Serial.println(")");
17.             }
18.          }
19.          else {
20.             Serial.print("*** ERROR - COMMAND ID not valid (");

21.             Serial.print(command[0], HEX);
22.             Serial.print(" ");
23.             Serial.print(command[1], HEX);
24.             Serial.println(")");
```

Figura 80. Función loop opción 3 y 4

En el fragmento de código mostrado en la Figura 80 se indica el procesamiento de las opciones “3” y “4”. Estas opciones son utilizadas para habilitar la recepción automática de los datos del vehículo previamente interrogados.

En caso de seleccionar la opción “3”, el sistema habilitará el *timer* definido que, realizando llamadas a la función *“sendPID()”*, enviará vía BLE la información actualizada. Si por el contrario se selecciona la opción “4”, el sistema procederá a detener el envío automático de datos al terminal móvil.

Existe una opción adicional añadida para que, en caso de no escoger una opción válida, el sistema muestre un mensaje de error vía serie.

4.4.3 Función setup()

En esta sección se analizará el fragmento de código indicado en la Figura 81, asociado a la función de inicialización del *firmware* correspondiente al dispositivo *RedBear Duo* en la plataforma HW/SW final.

```
1. void setup() {
2.   carloop.begin();
3.   Serial.begin(9600);
4.   delay(2000);
5.   Serial.println("BLE Carloop peripheral");
6.
7.   // Initialize ble_stack.
8.   ble.init();
9.
10.        // Register BLE callback functions.
11.        ble.onConnectedCallback(deviceConnectedCallback);
12.        ble.onDisconnectedCallback(deviceDisconnectedCallback);
13.        ble.onDataReadCallback(gattReadCallback);
14.        ble.onDataWriteCallback(gattWriteCallback);
15.
16.        // Add GAP service and characteristics
17.        ble.addService(BLE_UUID_GAP);
18.
   ble.addCharacteristic(BLE_UUID_GAP_CHARACTERISTIC_DEVICE_NAME, ATT_
   PROPERTY_READ|ATT_PROPERTY_WRITE, (uint8_t*)BLE_DEVICE_NAME, sizeof
   (BLE_DEVICE_NAME));
19.
   ble.addCharacteristic(BLE_UUID_GAP_CHARACTERISTIC_APPEARANCE, ATT_P
   ROPERTY_READ, appearance, sizeof(appearance));
20.
   ble.addCharacteristic(BLE_UUID_GAP_CHARACTERISTIC_PPCP, ATT_PROPERT
   Y_READ, conn_param, sizeof(conn_param));
21.
22.        // Add GATT service and characteristics
23.        ble.addService(BLE_UUID_GATT);
```

```
24.
   ble.addCharacteristic(BLE_UUID_GATT_CHARACTERISTIC_SERVICE_CHANGED,
    ATT_PROPERTY_INDICATE, change, sizeof(change));
25.
26.         // Add primary service1.
27.         ble.addService(service1_uuid);
28.         // Add characteristic to service1, return value handle of
   characteristic.
29.
   character1_handle = ble.addCharacteristicDynamic(char1_uuid, ATT_PR
   OPERTY_READ|ATT_PROPERTY_WRITE, characteristic1_data, CHARACTERISTI
   C1_MAX_LEN);
30.
   character2_handle = ble.addCharacteristicDynamic(char2_uuid, ATT_PR
   OPERTY_READ|ATT_PROPERTY_NOTIFY, characteristic2_data, CHARACTERIST
   IC2_MAX_LEN);
31.
32.
33.         // Set BLE advertising parameters
34.         ble.setAdvertisementParams(&adv_params);
35.
36.         // Set BLE advertising and scan respond data
37.         ble.setAdvertisementData(sizeof(adv_data), adv_data);
38.
   ble.setScanResponseData(sizeof(scan_response), scan_response);
39.
40.         // Start advertising.
41.         ble.startAdvertising();
42.         Serial.println("BLE Carloop peripheral - Start
   advertising");
43.       }
```

Figura 81. Función setup()

En esta sección de código se analiza la función *"setup()"*, que es la encargada de realizar la inicialización de los diferentes objetos y funciones necesarias para el correcto funcionamiento del código.

En primer lugar, se inicializa el objeto de tipo *"carloop"*, así como la interfaz serie que permite la comunicación con el ordenador portátil vía USB. Posteriormente, se muestra un mensaje vía serie que sirve de confirmación.

Una vez se hayan creado los objetos mencionados, se procede a inicializar el protocolo BLE en el rol de *Peripheral*. Para esto se hace uso de las funciones de las que dispone en el *firmware* propio el dispositivo *RedBear Duo.*

La primera función de inicialización es *"ble.init()"*, que debe ser llamada en primer lugar, ya que creará un hilo de ejecución encargado de gestionar los comandos y eventos asociados a la comunicación BLE.

Después, se procede a registrar las funciones de *callback* analizadas en el capítulo 4.3.1 “Funciones de comunicación”. De la misma forma, se definen los servicios y características que se van a implementar en el perfil GAP y GATT.

En este TFG se han utilizado las características incluidas en el servicio primario. La primera característica se caracteriza por disponer de las propiedades de lectura y escritura. En esta característica *char1* será en la que el usuario del sistema escriba la información en la que se encuentra el comando que desea ejecutar, de acuerdo con la nomenclatura vista en el apartado 4.4.3 “Función loop()”. La segunda característica, en cambio, será la encargada de recibir la información proveniente del vehículo y almacenarla para su lectura. En consecuencia, esta segunda característica *char2* dispone de las propiedades de lectura y notificación. Esto es debido a que el usuario no debe modificar el valor de esta característica, solo disponer de la capacidad de recepción de notificaciones y consultar la información recibida.

Caracterizados los perfiles GATT y GAP, se procede a definir los parámetros que se utilizarán en el proceso de *Advertising* para, posteriormente, ejecutar la función de *Advertising* del dispositivo *Peripheral* y mostrar un mensaje vía puerto serie confirmando su ejecución.

4.5 Corrección y validación

El procedimiento de comprobación del correcto funcionamiento de la plataforma final HW/SW desarrollada en este TFG, se basó en la verificación de los datos recibidos en el terminal móvil mediante la ayuda del software *nRF Connect* que, instalado en el terminal móvil con sistema operativo Android, servirá para identificar las tramas BLE recibidas desde el dispositivo *RedBear Duo*.

En la Figura 82 se muestra la interfaz del software *nRF Connect* una vez se ha puesto en funcionamiento la plataforma HW/SW, donde se observa que el dispositivo *Central* recibe correctamente los paquetes de *Advertising* enviados desde el dispositivo *RedBear Duo*, con los parámetros definidos en el *firmware*. Así, se ha identificado con el nombre “CARLOOP” y se proporciona una descripción de las características del dispositivo como la potencia de la señal, latencia, o su dirección MAC. A continuación, una vez establecida la

conexión BLE entre el terminal móvil y el dispositivo *RedBear Duo*, se procederá a comprobar el correcto funcionamiento de las funciones principales que rigen el comportamiento de la plataforma en su conjunto. Estas funciones son las dos principales analizadas en las versiones anteriores desarrolladas:

- Descubrir los PID disponibles para interrogar.
- Realizar peticiones de información a los PID que el usuario introduzca.

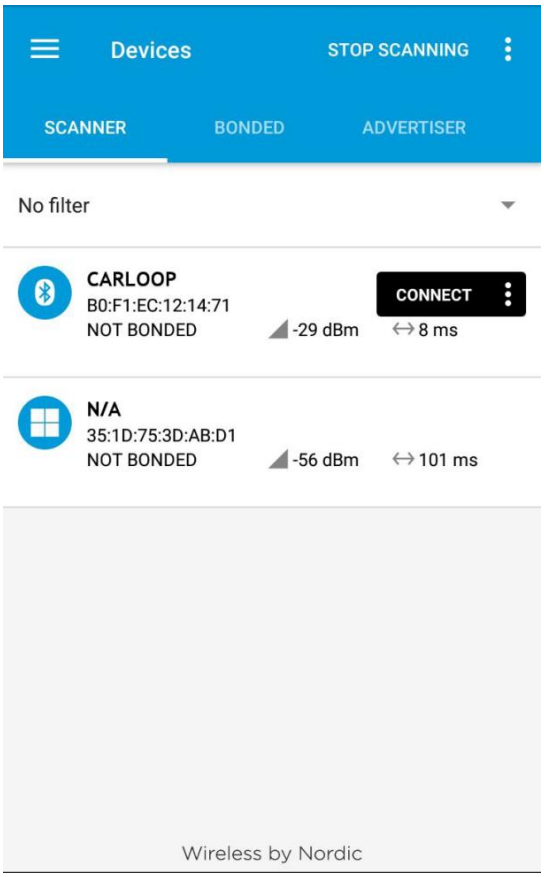

Figura 82. Puesta en marcha de la plataforma

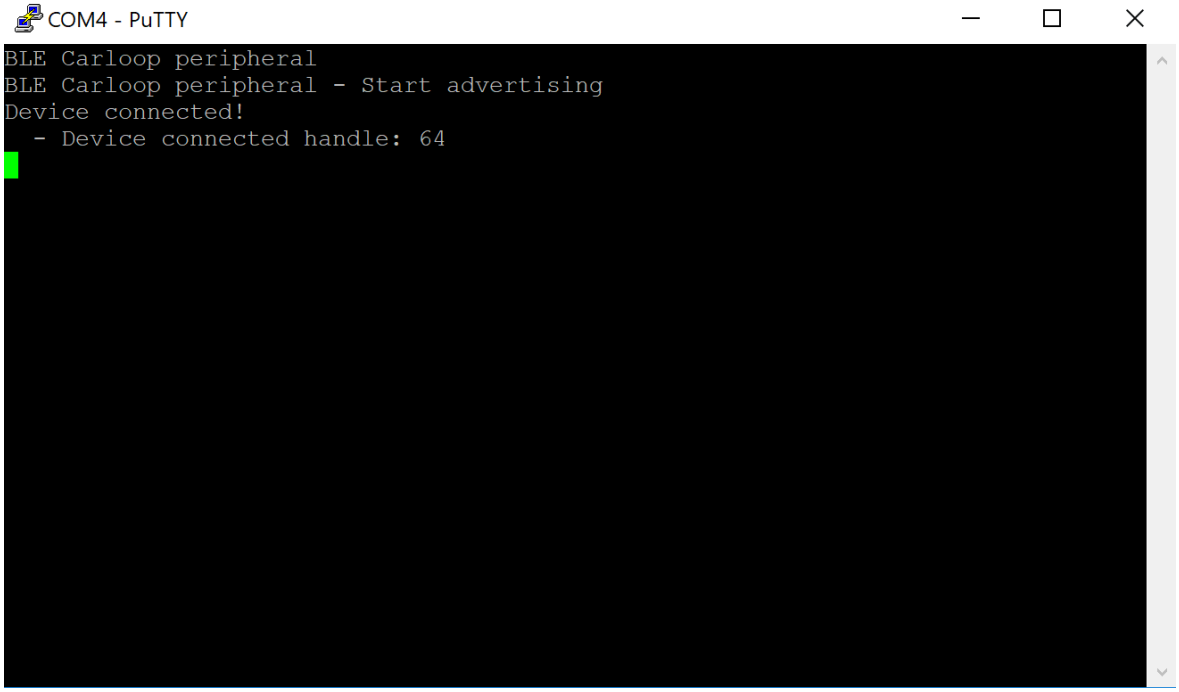

Figura 83.información vía serie del proceso de conexión

Por su parte, en el ordenador portátil se dispone de la interfaz serie habilitada para facilitar las tareas de corrección del código. En la Figura 83 se indica la información recibida por el dispositivo *RedBear Duo* una vez que se ha iniciado su funcionamiento. El dispositivo comienza a realizar el proceso de *Advertising* previsto hacia los dispositivos a su alcance.

En este caso, el usuario ha pulsado el botón *"connect"*, disponible en la interfaz mostrada en la Figura 84, y se ha establecido la comunicación BLE entre el dispositivo *RedBear Duo* y el terminal móvil. A continuación, se procederá a analizar la vista del usuario una vez establecida la comunicación con el dispositivo *RedBear Duo*.

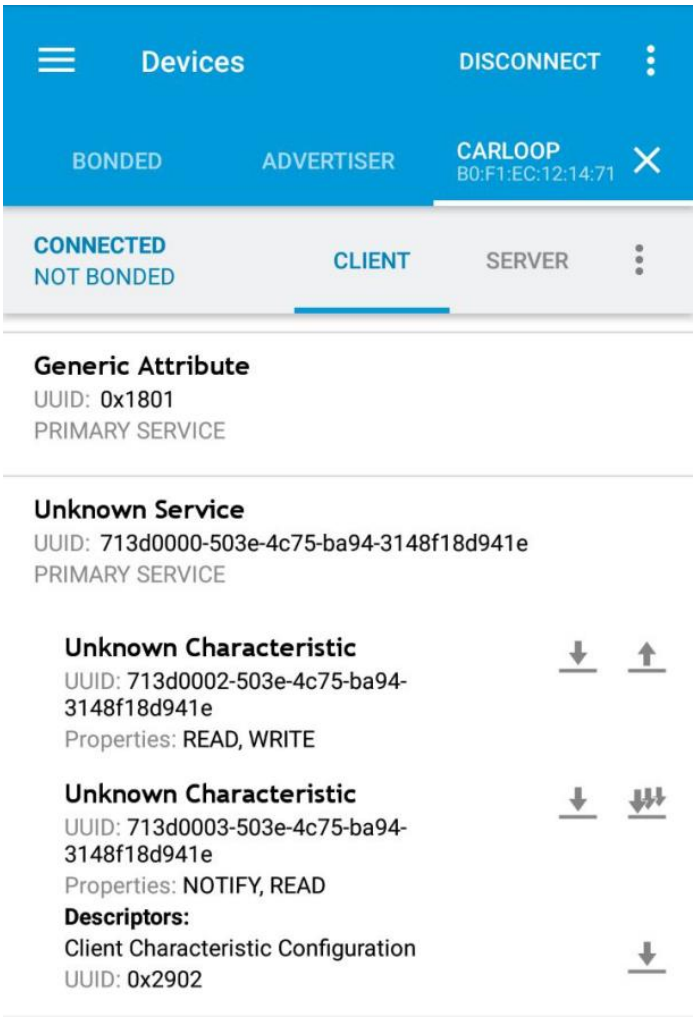

Figura 84. Interfaz software nRF Connect

Se puede apreciar la existencia de un atributo genérico y un servicio desconocido. Este último es el que se ha utilizado para almacenar las características que contienen la información. Tanto el atributo genérico como el servicio desconocido, y sus características asociadas, disponen de su propia identificación (UUID). Además, las características disponen de los atributos analizados en la sección 4.3 “Desarrollo del firmware”.

La primera característica es la utilizada para realizar los envíos de comandos hacia el dispositivo *RedBear Duo*, por lo que se puede leer y modificar la información que está almacenada en la característica. La segunda característica es la encargada de almacenar la información procedente del vehículo, por tanto, dispone de las propiedades de lectura y notificación.

4.5.1 Descubrir los PID disponibles

Esta función asociada al funcionamiento de la plataforma HW/SW final, se basa en el envío, vía BLE, desde el dispositivo móvil, de un comando definido para recibir la información de disponibilidad de las direcciones PID del vehículo.

Una vez que se ha inicializado la plataforma, y se ha establecido la conexión del dispositivo móvil con el dispositivo IoT *RedBearDuo*, se procede a enviar el primer comando.

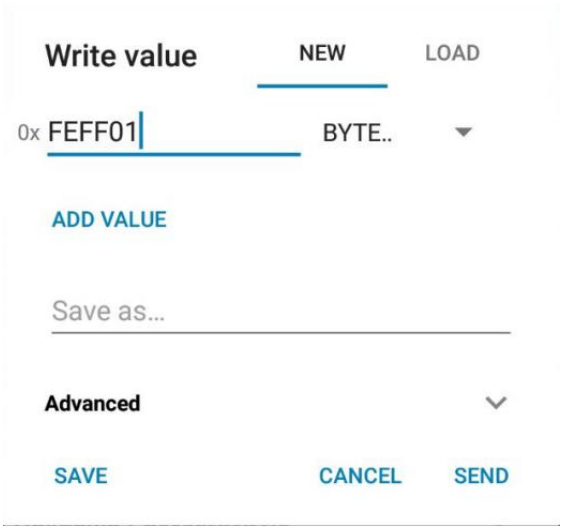

Figura 85. Comando para descubrir los PIDs disponibles

En la Figura 85 se aprecia la escritura en la primera característica de la cadena 0xFEFF01. Esa cadena corresponde, por orden, con:

- **FEFF:** Cabecera de dos *bytes* que se ha utilizado para la trama de recepción en el dispositivo *RedBear Duo.*
- **01:** Opción seleccionada por el usuario. Corresponde con la función de descubrir la disponibilidad de direcciones PID.

Una vez realizada la petición al dispositivo *RedBear Duo*, se muestra en la pantalla información indicada en la Figura 86.

Figura 86. Información de las características nRF Connect

En la Figura 86 se aprecia la escritura en la primera característica del comando anteriormente mencionado 0xFEFF01, así como la activación de las notificaciones en la segunda característica. En la segunda característica se aprecia que el valor es ahora 0xBE3EA813A0A007F011FEFED0. En este valor se incluye toda la información de disponibilidad que tiene el vehículo de prueba, del cual cada agrupación de cuatro *bytes* representa un grupo de disponibilidad de los que se han analizado.

Para continuar con la verificación de estos PID de disponibilidad, se procederá a realizar la comprobación de la función encargada de recibir la información desde el vehículo, donde se verificará la veracidad de la información obtenida acerca de la disponibilidad de PID.

4.5.2 Realizar peticiones a PID seleccionados.

En esta sección se comprobará, en primer lugar, el funcionamiento de la transmisión de información realizando una petición simple al PID encargado de devolver las revoluciones actuales del motor. Una vez comprobada la integridad de la información en las transmisiones, se procederá a realizar la prueba del peor caso posible, es decir, se realizará una petición a través del terminal móvil de todos los PID supuestamente disponibles del primer grupo de disponibilidad, según los resultados proporcionados por la rutina encargada de analizar la disponibilidad de PID.

Como se ha comentado, inicialmente se realizará una petición a un PID único a fin de comprobar la integridad de la información. En la Figura 87 se aprecia la visualización en el portátil de la rutina que se encarga de realizar las peticiones.

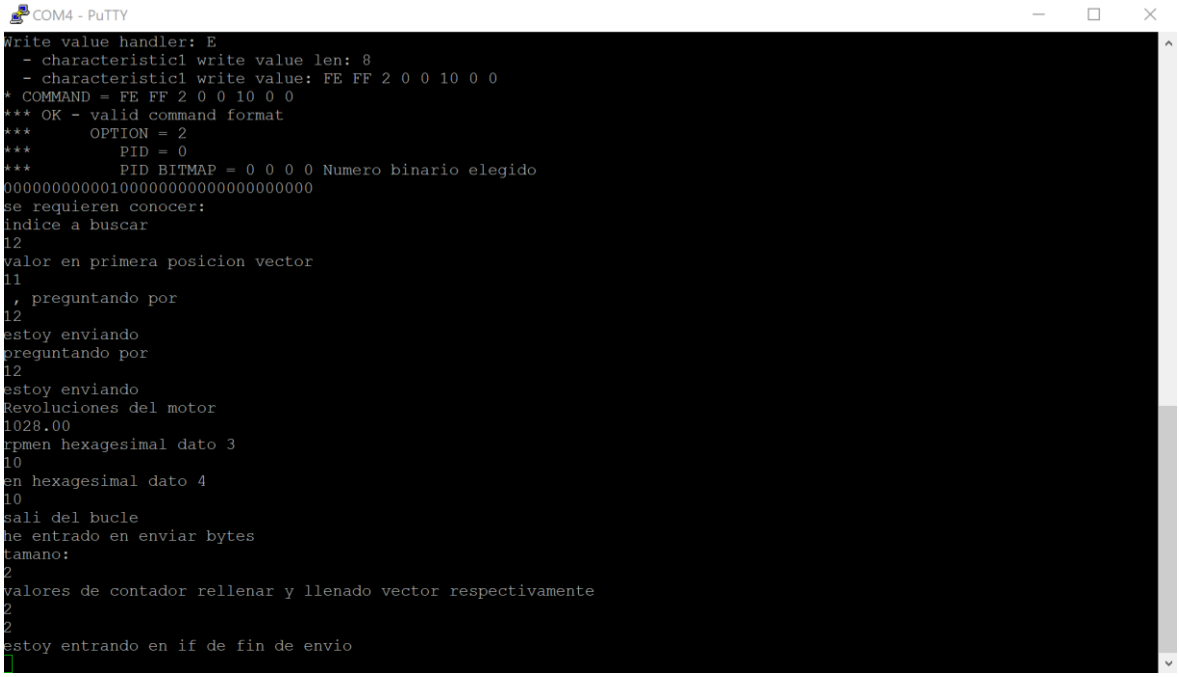

Figura 87. Puerto serie ante petición

Para realizar la petición se ha enviado al dispositivo el comando 0xFEFF020000100000, que hace referencia, en orden, a:

- **FEFF:** Cabecera de inicio.
- **02:** Función deseada -2-.
- **00:** Grupo de PID al que hace referencia -1º-.
- **00100000:** *Bitmap* de 4 *bytes* o 32 *bits* en el que se ha introducido un 1 en el PID asociado a las revoluciones del motor (0x0C o 12 en decimal).

En la Figura 87 se aprecia el análisis de los datos realizado por parte del dispositivo *RedBear Duo* que, en primera instancia, averigua a qué PID es al que hay que realizar la petición. Una vez reconocida la petición, se realiza y se obtiene la información procedente del vehículo. A continuación, se procede a analizar la longitud de la información procedente del PID asociado a las revoluciones del vehículo (en este caso 2 *bytes)* para así posicionar en orden los datos en la trama de datos limitada a 20 *bytes* que se enviará, vía BLE, al dispositivo móvil.

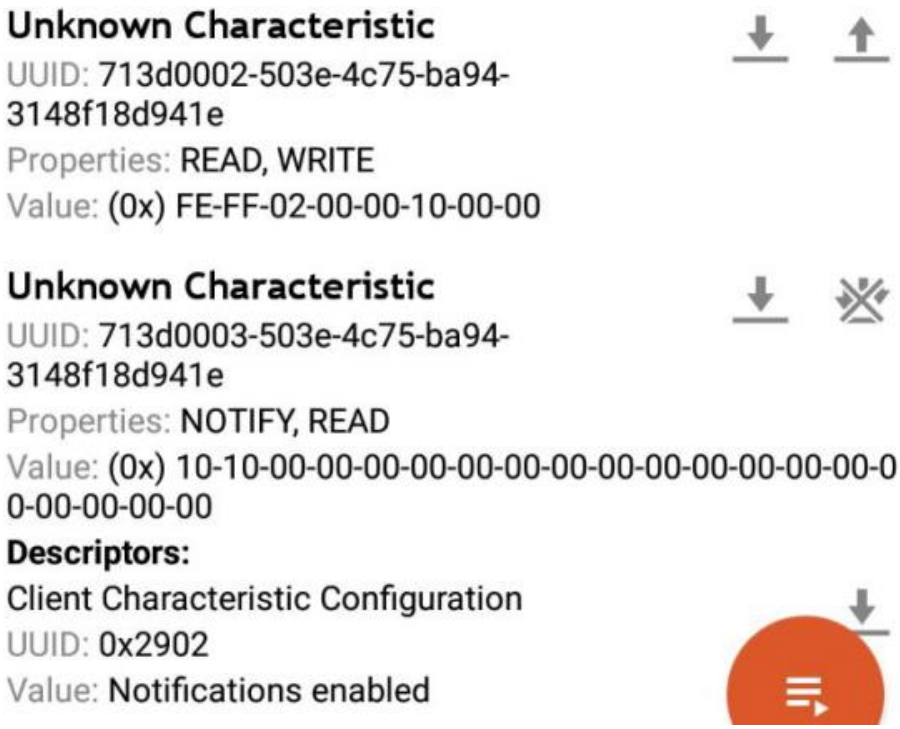

Figura 88. Información recibida en el terminal móvil

En la Figura 88 se indica la respuesta recibida ante la orden enviada. Se puede apreciar que la respuesta recibida, almacenada en la segunda característica, se corresponde con la información mostrada en la Figura 88 (0x1010), que representa un valor de 1028 RPM. Esta información fue fácilmente verificada, ya que en el momento de la prueba el vehículo estaba en ralentí (900-1100 RPM).

Ahora se procederá a realizar la prueba que comprobará el correcto funcionamiento del sistema, ya que se realizará una petición de todos los PID disponibles del primer grupo, según la información arrojada al realizar la petición de disponibilidad.

El conjunto de los PID agrupados en el primer grupo de PID en el vehículo, según la información de disponibilidad, es en total de 23 *bytes.* Estos 23 *bytes* exceden el límite de 20 *bytes* de datos asociado a la trama BLE, por lo que el sistema tendrá que actuar en consecuencia.

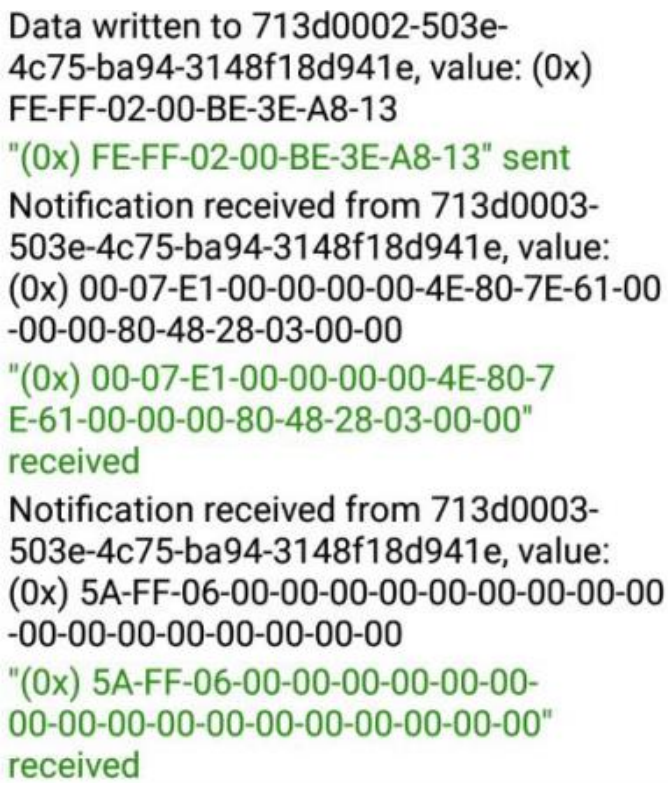

Figura 89. Envío y recepción

En la Figura 89 se indican las tramas enviadas y recibidas desde el dispositivo *RedBear Duo*. Se aprecia la escritura en la primera característica de la cadena del valor 0xFEFF0200BE3EA813, que corresponde a:

- FEFF: Cabecera de la trama.
- 02: Función elegida -2-.
- 00: Grupo de PID al que hace referencia -1º-.
- BE3EA813: Cadena binaria correspondiente a la respuesta recibida por la función de análisis de disponibilidad.

La respuesta recibida inicialmente se corresponde con los primeros 20 *bytes* de datos devueltos. Como se ha mencionado anteriormente, se esperan 23 *bytes* de datos provenientes del vehículo, por lo que no cabría en una única trama. Se aprecia en la Figura 89 la recepción de la primera trama de 20 *bytes*, y posteriormente se recibe otra trama de únicamente 3 *bytes,* que se corresponden con los que no cabían en la primera trama.

Después del análisis realizado, se puede concluir que el funcionamiento de la pasarela HW/SW final desarrollada en el presente TFG, es correcto.

5 Conclusiones

5.1 Conclusiones

Después de realizar el análisis de los resultados obtenidos a partir del proceso de validación funcional, se puede afirmar que se han cumplido los objetivos principales de este Trabajo Fin de Grado, habiéndose utilizado una estrategia claramente enfocada a la realización de la pasarela HW/SW mediante la separación de las tareas reconocidas.

En primer lugar se ha hecho un estudio de las características propias del bus de comunicación que utilizan los vehículos en la actualidad, ofreciendo una visión general de su posible interacción con la interfaz *carloop* que se ha utilizado para conectar los dispositivos IoT al conector de tipo OBD-II del vehículo de pruebas.

Después del análisis del contexto, se ha diseñado una pasarela inicial utilizando el dispositivo *Photon*, que carece de conectividad BLE, pero proporciona un punto de partida para la creación del *firmware* que se utilizará en la plataforma HW/SW final. Una vez que se comprobó la correcta interacción del bus de comunicación y el dispositivo *Photon*, utilizando la interfaz *carloop,* se procedió a codificar las funciones esenciales que permitirán al usuario conocer qué direcciones PID están disponibles y realizar consultas a éstas.

Estas funciones implementadas representaron el marco ideal para proceder a añadir la conectividad BLE de la que dispone el dispositivo *RedBear Duo.* Se ha realizado un análisis de la teoría relacionada con la tecnología BLE para caracterizar de manera correcta el código incluido en el *firmware* del dispositivo *RedBear Duo,* y así permitir que un usuario, mediante su terminal móvil, pueda solicitar y recibir la información proporcionada por la centralita del vehículo.

Desde una visión genérica, el estudio realizado en el desarrollo de este Trabajo Fin de Grado ha llevado consigo un aprendizaje ligado a la importancia de utilizar un bus de comunicación estándar, como es CAN, dentro de los vehículos de consumo en lugar de utilizar las alternativas propias de cada fabricante para implementarlas en su propio vehículo.

Este aspecto, sumado a la proliferación, por motivos ecológicos, de vehículos de carácter híbrido y eléctrico, en los cuales se hace patente el aumento de dispositivos electrónicos, hace que tecnologías como BLE supongan un avance en términos de practicidad, coste y escalabilidad.

También es remarcable la diferenciación del código que existe entre la primera implementación de la plataforma HW/SW mediante el dispositivo *Photon*, y el código utilizado en la última versión del *firmware* del dispositivo *RedBear Duo*, ya que en la última versión se observa una manera más eficiente de procesar los datos, que corresponde a la adquisición de experiencia por parte del estudiante.

No se ha mostrado la validación del envío periódico de datos debido a la imposibilidad de plantear una prueba capaz de verificar la recepción correcta de parámetros, ya que la poca o nula variación de la información actualizada del vehículo, no constituye una prueba de funcionamiento clara.

El *firmware* desarrollado para la elaboración del presente TFG se ha realizado de forma que esté capacitado para interactuar con cualquier vehículo fabricado a partir del año 2008 con cualquier tipo de disponibilidad de PID, siendo la plataforma HW/SW desarrollada la encargada de adaptarse y operar en consecuencia.

El estudio de la tecnología BLE, junto con la implementación de ésta en dispositivos IoT, ha creado una idea fundada de las capacidades disponibles en el campo del automovilismo para posibles integraciones con esta tecnología inalámbrica.

Por último, este trabajo representa una forma más de las posibilidades que representan los pequeños dispositivos inteligentes que, utilizados en combinación con diferentes tecnologías inalámbricas, permiten crear sistemas baratos, eficientes y con una escalabilidad muy alta.

5.2 Líneas Futuras

Este trabajo presenta las posibilidades de implementación que ofrece la tecnología BLE unida a dispositivos IoT. En él, se realiza el envío de información mediante la plataforma HW/SW desarrollada, teniendo esta información que se transmite, la característica de ser información en crudo. En consecuencia, una posible implementación futura podría ser la creación de una aplicación específica para el terminal móvil, capacitada para realizar la decodificación de la información recibida desde el vehículo.

Además de la implementación realizada, el bajo coste y la alta eficiencia permiten ampliar la visión mediante las siguientes propuestas de líneas futuras:

- Implementación de dispositivos inteligentes en un vehículo para utilizar funciones mediante un terminal móvil.
- Recogida de datos acerca del uso de un vehículo, orientada a *Big Data*.
- Redes de sensores de bajo coste basadas en tecnología BLE.

6 Bibliografía

[1] D. Evans and Cisco ISBG Group, "The Internet of Things How the Next Evolution of the Internet is Changing Everything," Abril 2011.

[2] "refrigerator", Samsung Electronics America, 2018. [Online]. Available: https://www.samsung.com/us/explore/family-hub-refrigerator/refrigerator/. [Accessed: 25- Jul- 2018].

[3] GrowthEnabler, "Market pulse report, internet of things (IoT)," Reino Unido, 2017.

[4] S. Corrigan, "Introduction to the controller area network (CAN)", Tech. Rep. SLOA101B, 2002.

[5] C. (CiA), "CAN in Automation (CiA): History of the CAN technology", Can-cia.org, 2018. [Online]. Available: https://www.can-cia.org/can-knowledge/can/can-history/. [Accessed: 25- Jul- 2018].

[6] C. Board, "Frequently Asked Questions (FAQ) About On-Board Diagnostic II (OBD II) Systems", Arb.ca.gov, 2018. [Online]. Available: https://www.arb.ca.gov/msprog/obdprog/obdfaq.htm. [Accessed: 25- Jul- 2018].

[7] P. David,"OBD II Diagnostic: Secrets Revealed", Florida: Kotzig Publishing, 2002.

[8] "definición y teoría del bus can", Catarina.udlap.mx, 2018. [Online]. Available: http://catarina.udlap.mx/u_dl_a/tales/documentos/lmt/pacheco_h_je/capitulo2.pdf. [Accessed: 26- Jul- 2018].

[9] C. Smith, "The Car Hacker's Handbook: A Guide for the Penetration Tester". (1st ed.) San Fancisco: No Starch Press, 2016.

[10] "On-board diagnostics", En.wikipedia.org, 2018. [Online]. Available: https://en.wikipedia.org/wiki/On-board_diagnostics. [Accessed: 26- Jul- 2018].

[11] "MacBook Pro (Retina, 15 pulgadas, finales de 2013) - Especificaciones técnicas", Support.apple.com, 2018. [Online]. Available: https://support.apple.com/kb/sp690?locale=es_ES. [Accessed: 26- Jul- 2018].

[12] "HUAWEI Mate 10 | Móvil Android | HUAWEI España", Consumer.huawei.com, 2018. [Online]. Available: https://consumer.huawei.com/es/phones/mate10/specs/. [Accessed: 26- Jul- 2018].

[13] "Hyundai ix35 Specifications | Hyundai Australia", Hyundai.com.au, 2018. [Online]. Available: https://www.hyundai.com.au/cars/suvs/ix35/specifications. [Accessed: 26- Jul- 2018].

[14] "Technical details of Photon". Available: https://store.particle.io/#photon. [Accessed: 26- Jul- 2018].

[15] “Technical details” of RedBear Duo. Available: https://redbear.cc/product/wifible/redbear-duo.html. [Accessed: 26- Jul- 2018]

[16] "Carloop Basic", Carloop, 2018. [Online]. Available: https://store.carloop.io/products/carloop-basic?variant=27724392961. [Accessed: 26- Jul- 2018].

[17] "Particle", Docs.particle.io, 2018. [Online]. Available: https://docs.particle.io/guide/getting-started/build/photon/. [Accessed: 26- Jul- 2018].

[18] "Getting Started with Carloop", Carloop.readme.io, 2018. [Online]. Available: https://carloop.readme.io/docs. [Accessed: 26- Jul- 2018].

[19] "Download PuTTY - a free SSH and telnet client for Windows", Putty.org, 2018. [Online]. Available: https://www.putty.org/. [Accessed: 26- Jul- 2018].

[20] "nRF Connect for Mobile / Nordic mobile Apps / Products / Home - Ultra Low Power Wireless Solutions from NORDIC SEMICONDUCTOR", Nordicsemi.com, 2018. [Online]. Available: https://www.nordicsemi.com/eng/Products/Nordic-mobile-Apps/nRF-Connect-for-Mobile. [Accessed: 26- Jul- 2018].

[21] "carloop/carloop-library", GitHub, 2018. [Online]. Available: https://github.com/carloop/carloop-library. [Accessed: 26- Jul- 2018].

[22]C. Gomez, J . Oller and Paradells Josep, "Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology," 2012.

[23] K. Cho et al, "Performance analysis of device discovery of Bluetooth Low Energy (BLE) networks," Computer Communications, vol. 81, pp. 72-85, 2016.

[24] M. Bhargava and J. Wilson,” IoT projects with Bluetooth low energy”.

[25] M. Bin Aftab, “Building bluetooth low energy systems”.

Pliego de condiciones

Se procede a presentar las condiciones bajo las que se ha desarrollado el presente Trabajo Fin de Grado. Se realizará una diferenciación entre el conjunto de herramientas *hardware*, *software* y *firmware* empleadas para llevar a cabo su realización.

Condiciones Hardware

En la Tabla 15 se indican el conjunto de dispositivos *hardware* utilizados, con su modelo correspondiente.

Tabla 15. Condiciones hardware.

Dispositivo/herramienta	Modelo	Fabricante/comerciante
Ordeador portátil	Macbook Pro 2013	Apple
Photon	Photon con pines soldados	Particle
RedBear Duo	RedBear Duo con pines soldados	RedBear
Kit de desarrollo Carloop	Interfaz carloop con conector OBDII hembra	Carloop
Coche de pruebas	Hyundai ix35 diésel	Hyundai
Teléfono móvil	Huawei Mate 10	Huawei

Condiciones Software

En la Tabla 16 se recoge las aplicaciones *software* utilizadas, con su versión correspondiente

Tabla 16. Condiciones software

Software	Versión	Desarrollador
Sistema operativo portátil	Microsoft Windows 7 Home Premium 64-bit Edition	Microsoft
Sistema operativo teléfono móvil	Android 8 Oreo	Huawei-Google
Microsoft Office	2016	Microsoft
Microsoft Visio	2016	Microsoft
Microsoft Project	2016	Microsoft

Particle IDE	-	Particle
PuTTY	V0.70	PuTTY project
Google Chrome	V67.0.3396.99/ 64 bit	Google
Adobe Reader	V11.0.21.18	Adobe Systems Software Ireland Ltd.
nRF Connect	V4.19.2	Nordic Semiconductor

Condiciones firmware

En la Tabla 17 se indica el *firmware* utilizado para cada dispositivo, y su versión correspondiente:

Tabla 17. Condiciones firmware

Firmware	Verisón
Photon	v0.6.2
RedBear Duo	v0.3.1

Presupuesto

En este capítulo se abordará el presupuesto que recoge los gastos generados en la realización del presente Trabajo Fin de Grado. El presupuesto, asimismo, está compuesto por las siguientes partes:

- Trabajo tarifado por tiempo empleado.
- Amortización del inmovilizado material, dividida a su vez en:
    - Amortización del material hardware.
    - Amortización del material software.
- Redacción de la documentación.
- Derechos de visado del COITT (Colegio Oficial de Ingenieros Técnicos de Telecomunicación).
- Gastos de tramitación y envío.
- Material fungible.

Una vez analizados los puntos que componen el presupuesto se procederá a aplicar los impuestos vigentes y se procederá a la obtención del coste total del Trabajo Fin de Grado.

Trabajo tarifado por tiempo empleado

En esta sección se contabilizarán los gastos que corresponden a la mano de obra según el salario correspondiente a la hora de trabajo de un Ingeniero Técnico de Telecomunicación. Con este objetivo se ha utilizado la fórmula de la Ecuación 1.

$$\boldsymbol{H = C_t \cdot 74,88 \cdot H_n + C_t \cdot 96,72 \cdot H_e} \qquad (1)$$

Donde:

- H: Honorarios totales por el tiempo dedicado.
- H_n: Número de horas normales trabajadas dentro de la jornada laboral.
- C_t: Factor de corrección que depende del número de horas trabajadas.
- H_e: Número de horas especiales trabajadas.

Se han invertido un total de 300 horas en la realización del presente Trabajo Fin de Grado. Todas ellas se han realizado dentro del horario normal, por lo que el número de horas especiales es cero. De acuerdo con lo establecido por el COITT, el factor de corrección C_t que se aplica para 300 horas trabajadas es de 0,60, como se aprecia en la Tabla 18.

Tabla 18. Coeficientes reductores para trabajo tarifado según el COITT

Horas	Factor de corrección
Hasta 36	1,00
Exceso de 36 hasta 72	0,90
Exceso de 72 hasta 108	0,80
Exceso de 108 hasta 144	0,70
Exceso de 144 hasta 180	0,65
Exceso de 180 hasta 360	0,60

Por tanto, haciendo uso de la ecuación 1:

$$\boldsymbol{H = 0,6 \cdot 74,88 \cdot 300 + 0,6 \cdot 96,72 \cdot 0 = 13.478,40€}$$

El trabajo tarifado por tiempo empleado asciende a la cantidad de ***trece mil cuatrocientos setenta y ocho euros con cuarenta céntimos.***

Amortización del inmovilizado material

El inmovilizado material se compone de los recursos *hardware* y *software* empleados para la realización de este Trabajo Fin de Grado.

Se ha utilizado un sistema de amortización lineal, en el que se supone que el inmovilizado material se deprecia de forma constante a lo largo de su vida útil. La cuota de amortización anual se calcula como se indica en la Ecuación 2.

$$\boldsymbol{Cuota\ anual = \frac{Valor\ de\ adquisición - Valor\ residual}{Número\ de\ años\ de\ vida\ útil}} \qquad (2)$$

El Valor residual corresponde con el valor teórico que se supone que tendrá el elemento en cuestión después de su vida útil. A continuación, se analizará la amortización de los recursos *hardware* y *software* utilizados.

Amortización del material *hardware*

En la Tabla 19 se especifican los elementos *hardware* amortizables utilizados para la realización del presente trabajo, indicando su valor de adquisición y su amortización.

Tabla 19. Amortización del material hardware

Dispositivo/Herramienta	Valor de adquisición	Valor de amortización
Ordenador Macbook Pro 2013	2380,00 €	119,00 €
Teléfono móvil Huawei Mate 10	590,00 €	103,25 €
Hyundai ix35	18.090,00 €	844,20 €
Kit de desarrollo Carloop + Photon	47,10 €	47,10 €
RedBear Duo	19,27 €	19,27 €
Total	21126,37 €	**1132,82 €**

Debido al bajo precio de los dispositivos, el conjunto formado por los dispositivos IoT, y el kit de desarrollo de *carloop*, su valor de amortización coincide con su valor de adquisición. Para el resto de elementos *hardware* se ha utilizado la formula indicada por la Ecuación 2.

El coste total del material hardware asciende a ***mil ciento treinta y dos euros con ochenta y dos céntimos.***

Amortización del material software

En la Tabla 20 se indican los elementos *software* amortizables utilizados para la realización del presente trabajo, indicando su valor de adquisición y su amortización.

Tabla 20. Amortización del software

Software	Valor de adquisición	Valor de amortización
Sistema operativo Windows 7 Home Premium	Licencia ULPGC	0,00 €
Particle IDE	0.00 € / Software acceso libre	0,00 €
Microsoft Visio	Licencia ULPGC	0,00 €
Microsoft Project	Licencia ULPGC	0,00 €
Microsoft Office	Licencia ULPGC	0,00 €

Google Chrome	0.00 € / Software acceso libre	0,00 €
Adobe Reader	0.00 € / Software acceso libre	0,00 €
PuTTY	0.00 € / Software acceso libre	0,00 €
Total	0,00 €	**0,00 €**

El coste total asociado al material *software* es de ***cero euros***.

Sumando los costes del inmovilizado del material *hardware* y *software* se obtiene el coste total de inmovilizado material, indicado en la Tabla 21.

Tabla 21. Coste total inmovilizado material

Concepto	Coste
Material hardware	1132,82 €
Material software	0,00 €
Total	**1132,82 €**

Por tanto, el coste total del inmovilizado material asciende a ***mil ciento treinta y dos euros con ochenta y dos céntimos.***

Redacción del trabajo

Para el cálculo del coste asociado a la redacción de la memoria del presente trabajo se ha usado la fórmula mostrada en la Ecuación 3.

$$\boldsymbol{R = 0,07 \cdot P \cdot C_n} \quad (3)$$

Donde:

- **R:** Honorarios por la redacción del trabajo.
- **P:** Presupuesto.
- $\boldsymbol{C_n}$: Coeficiente de ponderación en función del presupuesto.

El cálculo del presupuesto se realiza mediante la suman de los costes del trabajo tarifado por tiempo empleado y la amortización del inmovilizado material. En la Tabla 22 se indica este cálculo.

Tabla 22. Presupuesto

Concepto	Coste
Trabajo tarifado por tiempo empleado	13.478,40 €
Amortización del inmovilizado material	1132,82 €
Total	**14.611,22 €**

A continuación se analiza el coeficiente de ponderación C_n para este presupuesto. Según el COITT, para sueldos inferiores a 30.050,00 € les corresponde un valor de 1,00. En consecuencia, el coste asociado a la redacción del Trabajo Fin de Grado se indica en la Ecuación 4:

$$\boldsymbol{R = 0,07 \cdot 14.611,22 \cdot 1 = 1022,79\ €} \quad \textbf{(}4\text{)}$$

El coste de la redacción del trabajo asciende a ***mil veintidós euros con setenta y nueve céntimos***

Derechos de visado del COITT

Los derechos de visado para proyectos técnicos de carácter general en el año 2018 quedan establecidos por COITT, según la Ecuación 5.

$$\boldsymbol{V = 0,006 \cdot P_1 \cdot C_1 + 0,003 \cdot P_2 \cdot C_2} \quad (5)$$

Donde:

- **V:** Coste de visado del trabajo.
- $\boldsymbol{P_1}$: Presupuesto del proyecto.
- $\boldsymbol{C_1}$: Coeficiente reductor en función del presupuesto.
- $\boldsymbol{P_2}$: Presupuesto de ejecución material correspondiente a la obra civil.
- $\boldsymbol{C_2}$**:** Coeficiente reductor en función del presupuesto de ejecución material correspondiente a la obra civil.

El coeficiente C_1 es de 1,00 € para proyectos de presupuesto inferior a 30.050,00 €. Por otro lado, como El valor P_2 es de 0,00 € debido a que no se realiza ninguna obra civil, el coeficiente C_2 no se aplica. En la Tabla 23 se muestra el presupuesto del proyecto que se

obtiene al sumar las secciones correspondientes al trabajo tarifado por tiempo empleado, a la amortización del inmovilizado material y a la redacción del trabajo.

Tabla 23. Presupuesto incluyendo trabajo tarifado, amortización y coste de redacción

Concepto	Coste
Trabajo tarifado por tiempo empleado.	13.478,40 €
Amortización del inmovilizado material.	1132,82 €
Redacción del trabajo.	1022,79 €
Total	**15.634,01 €**

En consecuencia, el cálculo del coste por derechos de visados del presupuesto está indicado en la Ecuación 6.

$$\boldsymbol{V = 0,006 \cdot 15.634,01 \cdot 1 + 0,003 \cdot 0 \cdot \ C_2 = 93,80\ €} \quad (6)$$

Por tanto, el coste por derechos de visado del presupuesto asciende a ***noventa y tres euros con ochenta céntimos.***

Gastos de tramitación y envío

Los gastos derivados de la tramitación y envío ascienden a *seis* euros (6,00 €) por cada documento visado de forma telemática.

Material fungible

Durante el desarrollo de este Trabajo Fin de Grado se han empleado otros materiales aparte de los recursos *hardware* y *software* ya analizados. Estos materiales se documentan como material fungible. En la Tabla 24 se indican los costes derivados de estos recursos:

Tabla 24. Coste material fungible

Concepto	Coste
Folios	10,00 €
Tóner de la impresora	30,00 €
Encuadernado	5,00 €

Tres unidades CD-ROM	6,00 €
Total	**51,00** €

El coste del material fungible asciende a ***cincuenta y un euros.***

Aplicación de impuestos y coste total

La realización del presente Trabajo Fin de Grado está gravada con el Impuesto General Indirecto Canario (IGIC) en un siete por ciento (7%). En la Tabla 25 se indica el cálculo del presupuesto total del Trabajo Fin de Grado aplicando el impuesto.

Tabla 25. Presupuesto total Trabajo Fin de Grado

Concepto	Coste
Trabajo tarifado por tiempo empleado	13.478,40 €
Amortización del inmovilizado material	1132,82 €
Redacción del trabajo	1022,79 €
Derechos de visado del COITT	93,80 €
Gastos de tramitación y envío	6,00 €
Costes de material fungible	51,00 €
Total (sin IGIC)	**15.784,81 €**
IGIC (7%)	1104,94 €
Total	**16.889,75 €**

Por tanto, el presupuesto total del Trabajo Fin de Grado "Desarrollo de una plataforma para la interacción con la interfaz OBD-II utilizando BLE" asciende a ***dieciséis mil ochocientos ochenta y nueve euros con setenta y cinco céntimos***.

En Las Palmas de Gran Canaria, a 23 de julio de 2018.

Fdo. Cristóbal Macías Cabrera

Anexo I Decodificación respuesta PID

Primer bitmap [0x00 - 0x1F]

Tabla 26. Formulario de decodificación (primer bitmap)

PID (HEX)	Número de Bytes	Fórmula	Unidades
00	4	Apartado 2.2.3	-
01	4	Tabla 8	-
02	2	-	-
03	2	Tabla 11	-
04	1	$\frac{A}{2.55}$	%
05	1	A - 40	ºC
06	1	$\frac{A}{1.28}$ - 100	%
07	1		
08	1		
09	1		
0A	1	3A	KPa
0B	1	A	kPa
0C	2	$\frac{256A+B}{4}$	rpm
0D	1	A	Km/h
0E	1	$\frac{A}{2}$ - 64	Grados antes del punto muerto superior
0F	1	A - 40	ºC
10	2	$\frac{256A+B}{4}$	Gramos/seg
11	1	$\frac{A}{2.55}$	%
12	1	Tabla 12	-
13	1	Tabla 13	-
14	2	$\frac{A}{200}$ $\frac{B}{1.28}$ - 100	Voltios %
15	2		
16	2		
17	2		

18	2		
19	2		
1A	2		
1B	2		
1C	1	Tabla 14	-
1D	1	Tabla 15	-
1E	1	Tabla 16	-
1F	2	256A + B	segundos

Codificación a nivel de bit del primer bitmap

La codificación de la dirección PID 0x00 es la recogida en el apartado 2.2.3. Muestra las direcciones PID disponibles del primer *bitmap* de 32 bits.

PID 0x01

Tabla 27. Codificación a nivel de bit, PID 0x01

Bit	Nombre	Descripción
A7	MIL	Indica si la luz de mal función está en encendida.
A6-A0	DTC_CNT	Número de códigos DTC listos para mostrar.
B7	Reservado	-
B3	Sin nombre	0 -> Soporte para monitorización de ignición por chispa. (Gasolina) 1 ->Soporte para monitorización de ignición por compresión. (Diésel)

Al realizar una petición a esta dirección PID, el vehículo envía como respuesta 4 bytes de datos. El primer byte (A) es siempre cero, siendo los otros tres bytes (B, C, D) los que transmiten la información. El byte B indica la posibilidad de realizar diferentes tests al vehículo, denotando la posibilidad de realizar el test y/o avisar de un test que no se ha realizado con éxito. Los bytes C y D son dependientes del sistema de funcionamiento del motor, ya sea por ignición (gasolina) o compresión (Diésel).

En caso de que el motor del vehículo sea de ignición (Gasolina) los bytes C y D representan la información recogida en la Tabla 28.

Tabla 28. Codificación a nivel de bit, PID 0x41 (ignición)

	Test disponible	Test incompleto
Sistema EGR	C7	D7
Calentador del sensor de oxígeno	C6	D6
Sensor de oxígeno	C5	D5
Refrigerante aire acondicionado	C4	D4
Sistema de aire secundario	C3	D3
Sistema de evaporación	C2	D2
Catalizador a alta temperatura	C1	D1
Catalizador	C0	D0

En cambio, si el motor del vehículo es de compresión (Diésel) los bytes C y D representan la información recogida en la Tabla 29.

Tabla 29.Codificación a nivel de bit, PID 0x41 (Compresión)

	Test disponible	Test incompleto
Sistema EGR o VVT	C7	D7
Monitorización del filtro antipartículas	C6	D6
Sensor de gases de escape	C5	D5
Reservado	C4	D4
Presión del turbo	C3	D3
Reservado	C2	D2
monitorización NOx	C1	D1
Catalizador de hidrocarburos no metánicos	C0	D0

PID 0x03

Tabla 30.Codificación a nivel de bit, PID 0x03

Valor	Descripción
1	Flujo de subida.
2	Flujo de bajada o conversión del catalizador.
4	Atmosférico o apagado.
8	Bomba encendida para diagnóstico.

PID 0x12

Tabla 31.Codificación a nivel de bit, PID 0x12

Valor	Descripción
1	Circuito abierto debido a temperatura del motor insuficiente.
2	Circuito cerrado, usando la información del sensor de oxígeno para calcular la mezcla.
4	Circuito abierto debido a motor en carga o corte de combustible debido a desaceleración.
8	Circuito abierto debido a fallo del sistema.
16	Circuito cerrado, utilizando al menos un sensor de oxígeno pero existe un error en el sistema de retroalimentado de los sensores de oxígeno.

PID 0x13

Tabla 32. Codificación a nivel de bit, PID 0x13

Banco nº	Sensores 1-4			
1	A0	A1	A2	A3
2	A4	A5	A6	A7

PID 0x1C

Tabla 33. Codificación a nivel de bit, PID 0x1C

Valor	Descripción
1	OBD-II definido por California Air Resources Board (CARB)
2	OBD definido por Environmental Protection Agency (EPA)

3	OBD y OBD-II
4	OBD-I
5	No OBD
6	Europe OBD
7	Europe OBD y OBD-II
8	Europe OBD y OBD.
9	Europe OBD, OBD y OBD-II
10	Japan OBD
11	Japan OBD y OBD-II
12	Japan OBD y Europe OBD
13	Japan OBD, Europe OBD y OBD-II
14	Reservado
15	Reservado
16	Reservado
17	Diagnosis del motor del fabricante
18	Diagnosis ampliada del motor del fabricante
19	Diagnóstico de vehículos de alta masa
20	Diagnóstico de vehículos de alta masa
21	OBD armonizado mundial
22	Reservado
23	Europe OBD para vehículos pesados sin control de NOx (fase 1)
24	Europe OBD para vehículos pesados con control de NOx (fase 1)
25	Europe OBD para vehículos pesados sin control de NOx (fase 2)
26	Europe OBD para vehículos pesados con control de NOx (fase 2)
27	Reservado
28	Brazil OBD (fase 1)
29	Brazil OBD (fase 2)
30	Korean OBD
31	India OBD-I
32	Indian OBD-II
33	Europe OBD para vehículos pesados (fase 4)
34-250	Reservado
251-255	No disponible para asignación

PID 0x1D

Tabla 34. Codificación a nivel de bit, PID 0x1D

Banco nº	Sensores 1-2	
1	A0	A1
2	A2	A3
3	A4	A5
4	A6	A7

PID 0x1E

Tabla 35.Codificación a nivel de bit PID 0x1E

Bit	Descripción
A0	Estado del sistema de alimentación de salida.
A1-A7	No se utiliza

Segundo bitmap [0x20 - 0x3F]

Tabla 36. Formulario de decodificación (segundo bitmap)

PID (HEX)	Número de Bytes	Fórmula	Unidades
20	4	Apartado 2.2.3	-
21	2	256A + B	km
22	2	0.079(256A + B)	kPa
23	2	10(256A + B)	kPa
24	4	$\frac{2}{65536}$(256A + B) $\frac{8}{65536}$(256C + D)	Proporción V
25	4		
26	4		
27	4		
28	4		
29	4		
2A	4		
2B	4		
2C	1	$\frac{A}{2.55}$	%

2D	1	$\frac{A}{1.28} - 100$	%
2E	1	$\frac{A}{2.55}$	%
2F	1	$\frac{A}{2.55}$	%
30	1	A	Contador
31	2	256A + B	%
32	2	$\frac{256A + B}{4}$	Pa
33	1	A	kPa
34	4	$\frac{2}{65536}$(256A + B) C + $\frac{D}{256}$ - 128	Proporción mA
35	4		
36	4		
37	4		
38	4		
39	4		
3A	4		
3B	4		
3C	2	$\frac{256A + B}{10}$ - 40	ºC
3D	2		
3E	2		
3F	2		

Codificación a nivel de bit del segundo bitmap

Cabe destacar que en esta sección no es necesario realizar un desglose a nivel de bit ya que no existen direcciones PID que estén codificadas de tal manera.

Tercer bitmap [0x40 - 0x5F]

Tabla 37. Formulario de decodificación (tercer bitmap)

PID (HEX)	Número de Bytes	Fórmula	Unidades
40	4	Apartado 2.2.3	-
41	4	Tabla 19	-

42	2	$\frac{256A + B}{1000}$	V
43	2	$\frac{256A + B}{2.55}$	%
44	2	$\frac{2}{65536}$(256A+B)	Proporción
45	1	$\frac{A}{2.55}$	%
46	1	A - 40	ºC
47	1	$\frac{A}{2.55}$	%
48	4		
49	4		
4A	4		
4B	4		
4C	1		
4D	2	$256A + B$	Minutos
4E	2		
4F	4	A, B, C, 10D	Proporción, V, mA, kPa
50	4	10A	Gramos/seg
51	1	-	%
52	1	$\frac{A}{2.55}$	%
53	2	$\frac{256A + B}{200}$	kPa
54	2	$(256A + B) - 32767$	Pa
55	2	$\frac{A}{1.28} - 100$	%
56	2		
57	2	$\frac{B}{1.28} - 100$	
58	2		
59	2	10(256A + B)	kPa
5A	1	$\frac{A}{2.55}$	%
5B	1	$\frac{A}{2.55}$	%
5C	1	A - 40	ºC
5D	2	$\frac{256A + B}{128} - 210$	º

5E	2	$\frac{256A + B}{20}$	Litros/hora
5F	1	256A + B	segundos

Codificación a nivel de bit del tercer bitmap

Tabla 38.Codificación a nivel de bit PID 0x41

	Test disponible	**Test incompleto**
Componentes	B2	B6
Sistema de combustible	B1	B5
Fallos	B0	B4

Si el vehículo es por ignición (gasolina) los bytes C y D representan:

Tabla 39. Codificación a nivel de bit, PID 0x41 (ignición)

	Test disponible	**Test incompleto**
Sistema EGR	C7	D7
Calentador del sensor de oxígeno	C6	D6
Sensor de oxígeno	C5	D5
Refrigerante aire acondicionado	C4	D4
Sistema de aire secundario	C3	D3
Sistema de evaporación	C2	D2
Catalizador a alta temperatura	C1	D1
Catalizador	C0	D0

Si en cambio, el vehículo es de compresión (Diésel):

Tabla 40.Codificación a nivel de bit, PID 0x41 (Compresión)

	Test disponible	**Test incompleto**
Sistema EGR o VVT	C7	D7

Monitorización del filtro antipartículas	C6	D6
Sensor de gases de escape	C5	D5
Reservado	C4	D4
Presión del turbo	C3	D3
Reservado	C2	D2
monitorización NOx	C1	D1
Catalizador de hidrocarburos no metánicos	C0	D0

Anexo II Contenido del CD-ROM

Adjunto a la memoria del presente Trabajo Fin de Grado se encuentra disponible un *Compact Disc Read-Only Memory (CD-ROM).* En este anexo se especifica el contenido incluido:

- En el directorio *Memoria Trabajo Fin de Grado* se encuentra la memoria del TFG *"Desarrollo de una plataforma para la interacción con la interfaz OBD-II utilizando BLE"* en lengua española y en formato PDF.